Revised
Includes Latest Information

The SOUND BLASTER™ Book

Join the SOUND EXPLOSION

Includes ready-to-use companion diskette

Axel Stolz

- Learn the elements of music and sound
- Discover and study the MIDI language
- Learn about sound sampling, storing and sound playback
- Understand SBI (Sound Blaster Instrument) format and FM synthesis
- Tips and tricks for using Sound Blaster and Sound Blaster Pro

Abacus
A Data Becker Book

What readers are saying about The Sound Blaster Book

"Good book." R.A. NY

"Very informative." M.M. D.E.

"Great book!" F.K.

"Great book. I always rely on Abacus to get information I can't get anywhere else." P.B. IL

"Excellent." S.K. AR

"Good book." T.J. FL

"A great book." L.R. MD

"Very good." B.W. NY

"A very informative book." K.S. CO

"Excellent book." G.E. PA

"Good book." D.R. LA

"Could have used this when I got the SB card!" F.M. MD

"Good... much cheaper than retail $100.00 manual." B.C. NY

"Great book...keep me updated on new versions of this book please." R.R. IA

"Wow!" O.B. WI

"Excellent book!" G.M. IL

"Good book!" S.S. CA

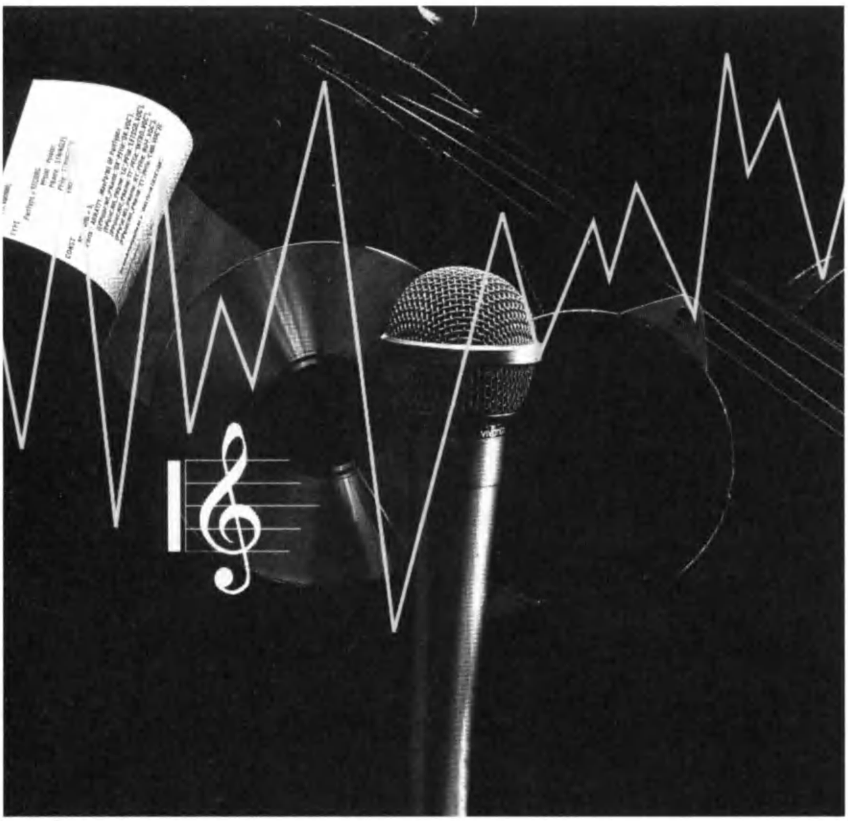
"Excellent book, answers all the questions raised by examining the Sound Blaster manuals." L.B. PA

"The included source code is an incredible help!" D.K, NY

"Good reference book for Sound Blaster card. Utilities disk looks good." F.R. MA

"I like the examples in Pascal." B.B. IN

The SOUND BLASTER™ Book



by Axel Stolz

Abacus
A Data Becker Book

Printed in U.S.A.

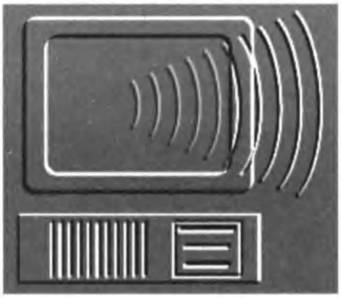
Managing Editor: Louise Benzer
Editors: Louise Benzer, Gene Traas, Robbin Markley
Technical Editors: Gene Traas
Book Design: Scott Slaughter
Cover Art: Dick Droste

Library of Congress Cataloging-in-Publication Data

```
Stolz, Axel, 1968-
    The Sound Blaster book / Axel Stolz.
       p.  cm.
    Includes index.
    ISBN 1-55755-181-2  :  $34.95
    1. Expansion boards (Microcomputers)   2. Computer sound
  processing.           I.  Title.
  TK7895.E96S76    1992
  006.5--dc20                                          92-30688
                                                            CIP
```

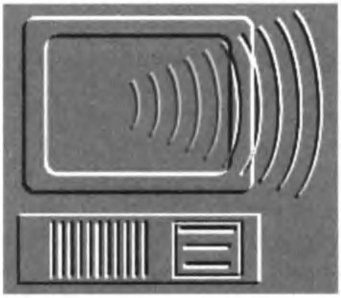
Printed in USA

10 9 8 7 6 5 4 3 2 1



The Sound Blaster Book

Contents



Chapter 1

PC Sound

Technological development has increased rapidly over the last few years. As a result, PC users have access to the computing power that was previously available only to professional programmers.

Today most personal computers include a standard VGA (Video Graphics Array) card for high-resolution color graphics display. Prices for these cards range from $50 to $350 for a card with 256K, 512K or even 1024K of video RAM. A VGA card enables PC users to add impressive graphics to their screen displays.

However, if you want "multimedia" realism on your system, you may feel that something is still missing, even with a VGA card. For example, when you boot your computer, you may be disappointed by the simple "beep" that you hear while the AUTOEXEC.BAT file executes.

So you must find a way to bridge the gap between the sound and graphics qualities of your personal computer.

A few years ago doing this was very costly. Early sound cards, which were designed to add music to computers, were very expensive. Even today specialized sound cards can cost $700 or more. Obviously this is too expensive for the average user. Also, these cards are usually designed for professional users.

IBM Music Feature Card

The IBM Music Feature card is one of the original sound cards. This fairly expensive card includes an eight-voice stereo synthesizer and a complete MIDI interface. The heart of this card is the Yamaha YM-2164 sound chip, which can also be found in the Yamaha FB-01 MIDI Expander.

Sound generation takes place through an FM synthesizer with multiple control parameters. There are also 240 preprogrammed sounds, including reproductions of traditional musical instruments.

A special feature of this card (although perhaps not too practical, considering its price—up to $800 each) is that up to four of them can be installed at a time, which allows up to 32 simultaneous voices.

Most of Sierra Online's games support the IBM Music Feature Card.

Roland LAPC-1

Another professional sound card is the Roland LAPC-1, which currently costs about $850. It is eight-voice polyphonic and comes with 128 preprogrammed sounds, which can also be modified. The LAPC-1 also provides a drum computer and digital effects device. Because of these features, this card is too powerful to be used for only games.

By using an add-on MIDI box, you can produce professional sound applications. However, remember that, in price and quality, the available software and hardware extensions dramatically increase this card's power.

Neither of these professional sound cards represents the latest in synthesizer technology. Although many computer musicians use these cards, eight-voice polyphony is no longer enough to fully use MIDI capabilities.

Beginning in 1987, some powerful but less expensive sound cards began competing with these professional cards. Many amateur musicians and computer game users began using these less expensive cards.

Game Blaster

The Game Blaster card was the first one of these cards to appear. It cost about $340 to $400. However, as an amateur sound card, Game Blaster was a little ahead of its time. So it wasn't very successful.

AdLib

The AdLib card was more successful. Its introduction coincided with the rapidly growing market for PC-based computer games. Software firms that had provided music for the Roland and IBM cards also began working with AdLib.

With this software support and its affordable price ($350 to $400), the AdLib card soon emerged as the standard for sound cards intended for non-professional users.

Sound Blaster

Sound Blaster, introduced in 1989, was AdLib's first competition. The Sound Blaster card had three features that weren't included in the other sound cards available at that time.

This card was fully compatible with Game Blaster and used all existing Game Blaster software. So new software development wasn't needed. Also, its manufacturer, Creative Labs (which had also produced Game Blaster), could continue to support its original product.

Also, the Sound Blaster card was completely compatible with the AdLib card. This opened up a broad software market for Sound Blaster and directly challenged the AdLib card.

The most important characteristic of this card, however, was its integrated sampling capability, with one digital audio channel for recording and playing nature sounds, music, or speech.

Competition between the two major manufacturers, AdLib and Creative Labs, rapidly decreased prices. Initially priced between $350 to $400, Sound Blaster now sells for $100 to $200. The AdLib card costs $170 to $230.

Because of these developments, the PC now has audio capabilities that are comparable to many other computers. So by combining sound and graphics, you can make your system come alive.

Different versions of Sound Blaster

Basic differences among Sound Blaster versions

After it was introduced, the Sound Blaster card was changed by a few important hardware developments. These developments help distinguish between Versions 1.0, 1.5, and 2.0.

Sound Blaster Pro is so innovative that it could be placed in its own category. The most recent version is 2.0.

Some technical terms, such as "sample" and "sampling rate," appearing in the following sections, will be explained in more detail later. These terms refer to the fundamentals of digital sound. You'll learn about them soon by performing a few experiments with Sound Blaster.

The first Sound Blaster card

Sound Blaster Version 1.0

Version 1.0 of Sound Blaster, which was introduced in 1989, was the first Sound Blaster card. Since this version contains the C/MS chips, it's compatible with Game Blaster. With Versions 1.5 and

2.0, these chips must be purchased separately. However, these chips aren't actually necessary.

Currently the C/MS chips aren't supported by many software programs. The game "Silpheed," from Sierra, provides an example of good C/MS-music. Otherwise, Game Blaster software is very rare.

When it was introduced, Game Blaster was a very versatile product. It provided the PC with 2 x 6 voice stereo sound. Unfortunately, the sounds themselves weren't that spectacular. This is because the sine-wave generator used to produce the sounds is capable of only low-quality, artificial-sounding output. FM voices, which are available in the AdLib card, provide much better instrumental music and special effects.

Version 1.0 includes 12 voice C/MS stereo sound, 11 voice FM mono sound, and one digital audio channel. It also provides a MIDI interface (accessible through optional MIDI connections) and a joystick connector.

Sampling rates for the digital audio channel range from 4 to 15 KHz for recording and 4 to 24 KHz for playback. With suitable software and a fast computer, these values can be improved slightly. However, the results achieved under normal conditions are capable of producing good digital PC sound.

Sound Blaster Version 1.0 Specifications	
Analog-Digital Input	4,000 - 23,000 Hz mono
Analog-Digital Output	4,000 - 23,000 Hz mono
Sample-Resolution	8-bit
Sample-Inputs	Mic mono
FM-Music	11 voice mono
CMS-Music	12 voice stereo
CD-ROM Connector	No
Amplifier	4 Watt / 4 Ohm

Sound Blaster Version 1.5

Eliminated C/MS chips

There are two basic differences between Version 1.5 and Version 1.0. The first difference is that the C/MS chips were eliminated in Version 1.5. This enabled Creative Labs to reduce the price and, therefore, expand its market.

Upgrade capability was built into Version 1.5 so the C/MS chips could be added, if desired. This produces the same hardware configuration as Version 1.0 and provides access to all Game Blaster software.

Without the upgrade, Version 1.5 includes 11 voice FM mono sound and one digital audio channel (identical to Version 1.0). Version 1.5 also provides a MIDI interface (accessible through optional MIDI connections) and a joystick connector.

Another important difference from Version 1.0 is the software. You'll have either Voice Editor I or SBTALKER/Dr. Sbaitso. We'll describe these programs later. Regardless of which package you have, Version 1.5 represents a significant improvement over Version 1.0.

Sound Blaster Version 1.5 Specifications	
Analog-Digital Input	4,000 - 23,000 Hz mono
Analog-Digital Output	4,000 - 23,000 Hz mono
Sample-Resolution	8-bit
Sample-Inputs	Mic mono
FM-Music	11 voice mono
CMS-Music	Upgradable
CD-ROM Connector	No
Amplifier	4 Watt / 4 Ohm

Sound Blaster Version 2.0

11 voice FM mono sound and one digital audio channel

Similar to Version 1.5, Version 2.0 can also be upgraded by adding the C/MS chip set. But other differences distinguish it from both Versions 1.0 and 1.5.

Without the C/MS upgrade, Version 2.0 includes 11 voice FM mono sound and one digital audio channel. However, this version has improved specifications for the digital audio channel.

The recording sampling rate still ranges from 4 to 15 KHz. The playback sampling rate, however, is 44.1 KHz, comparable to the sampling rate found on a CD player. This means that with Version 2.0 you can play exceptionally high-quality samples recorded using another card.

Another important feature of Version 2.0 is that it contains separate audio inputs for the microphone and the Line-In signal.

So you can have a microphone connected and at the same time use your stereo as the sound source for the Line-In jack. So you don't have to change connections continually.

The same software is used for both Version 2.0 and Version 1.5.

Sound Blaster Version 2.0 Specifications	
Analog-Digital Input	4,000 - 23,000 Hz mono
Analog-Digital Output	4,000 - 44,100 Hz mono
Sample-Resolution	8-bit
Sample-Inputs	Mic mono
	Line-In mono
FM-Music	11 voice mono
CMS-Music	Upgradable
CD-ROM Connector	No
Amplifier	4 Watt / 4 Ohm

Sound Blaster Pro 1.0

Significant improvement

Sound Blaster Pro represents a significant improvement over the earlier Sound Blaster versions. It's actually a completely different product, although it's compatible with the previous versions. However, C/MS capability is no longer supported.

The SB Pro requires at least a 286 AT or compatible computer, since it uses a 16-bit slot. However, the 8-bit cards of Versions 1.0, 1.5, and 2.0 can be used in XT and compatible machines.

Sound Blaster Pro is a professional-quality tool for multimedia applications. Despite its versatility and power, it's less expensive than the original Sound Blaster Version 1.0.

It features 2 x 11 voice FM stereo sound, one stereo digital audio channel, a MIDI interface, a joystick connector, and a connector for a CD-ROM drive. The FM voices are produced by double the number of FM chips found on earlier versions. So 11 voices per stereo channel can be created.

Digital audio channel

The digital audio channel has impressive capabilities. It can digitize sounds in stereo mode at sampling rates from 2 x 4 KHz up to 2 x 22.05 KHz, or in mono mode from 4 KHz up to 44.1 KHz. So you can achieve the same quality digital recording as a CD, except not in stereo. Obviously, sounds can be played back with the same quality (i.e., sampling rate) that was used to record them.

Possible sound sources

There are three possible sound sources (recording inputs) for Sound Blaster Pro. The first two, the microphone and the Line-In input, are the same as for Version 2.0. You can also connect a CD-ROM drive and sample directly from an audio CD. This method achieves the highest quality recording, although it uses analog data transfer instead of digital data transfer.

Another advantage of SB Pro is its ability to control the volumes of all internal or external sound sources by using software. A mixing program lets you set individual sources louder or softer as desired and also provides a master volume control.

These capabilities make Sound Blaster Pro the standard for the fast-growing multimedia field.

Sound Blaster Pro 1.0 Specifications	
Analog-Digital Input	4,000 - 44,100 Hz mono 4,000 - 22,050 Hz stereo
Analog-Digital Output mono/stereo	4,000 - 44,100 Hz
Sample-Resolution	8-bit
Sample-Inputs	Mic mono Line-In stereo CD stereo
FM-Music	22 voice stereo
CMS-Music	No
CD-ROM Connector	Yes
Amplifier	4 Watt / 4 Ohm

Sound Blaster Pro 2.0

20 FM stereo voices

The only difference between the newest version of Sound Blaster Pro (Version 2.0) and SB Pro 1.0 is the way FM sounds are generated. The old FM chips have been replaced with Yamaha OPL3 chips, which provide more operators for sound generation.

In FM synthesis, the more operators in use for generating and refining the sound, the better quality the sound. While the old chips used a maximum of two operators per voice, the OPL3 uses two or four. With these, Sound Blaster Pro 2.0 provides a total of 20 FM stereo voices.

Future software programs written for SB Pro 2.0 won't be compatible with SB Pro 1.0. So this will negatively affect SB Pro

1.0's value. Unfortunately, SB Pro 1.0 wasn't originally equipped with higher performance chips.

However, the trend is moving toward music over the digital audio channel. So Sound Blaster Pro 1.0 still represents the current state of the Sound Blaster family.

Sound Blaster Pro 2.0 Specifications	
Analog-Digital Input	4,000 - 44,100 Hz mono 4,000 - 22,050 Hz stereo
Analog-Digital Output	4,000 - 44,100 Hz mono/stereo
Sample-Resolution	8-bit
Sample-Inputs	Mic mono Line-In stereo CD stereo
FM-Music	22 voice stereo 20 voice OPL3 stereo
CMS-Music	No
CD-ROM Connector	Yes
Amplifier	4 Watt / 4 Ohm

Sound Blaster 16 ASP

The newest addition to the Sound Blaster family is Sound Blaster 16 ASP. This version offers 16-bit, CD-quality stereo sampling and playback. SB 16 ASP uses a programmable Advanced Signal Processor for high speed and digital signal processing, speech recognition, special effects and real-time hardware compression and decompression. This processor handles all digital audio processing so your CPU can work at full capacity.

Hardware requirements

Sound Blaster 16 ASP requires at least a 286 AT or compatible computer because it uses a 16-bit slot. It also requires at least 64K of RAM, and a VGA card is recommended.

Besides improved sound quality, SB 16 ASP offers numerous utilities for recording and manipulating sound. It also includes output mixing sources, such as a microphone, stereo line-in and FM synthesis, and a stereo digital mixer. Other features include a built-in MIDI interface, a CD-ROM interface, a joystick port, and an output power amplifier.

Sound Blaster 16 ASP also includes an optional expansion board, called Wave Blaster, that attaches directly to the SB 16 ASP.

This board produces almost perfect reproductions of instrument sounds during the playback of MIDI files. All sound samples are stored on-board and have full 16-bit resolution.

In addition to normal Sound Blaster MIDI support through the Creative Labs MIDI Kit, Sound Blaster 16 ASP features MPU-401 UART MIDI support. Roland Corporation's MPU-401 interfaces set the standard for PC MIDI interfacing, and the SB 16 ASP now supports the UART implementation of MPU-401 MIDI.

Several programs are also included with the Sound Blaster 16 ASP. There are two multimedia applications: HSC InterActive and PC Animate Plus. HSC InterActive is used for authoring multimedia applications and PC Animate Plus is used for creating animations.

There are also two programs for sampling in Windows. You can use WaveStudio for sampling and editing within the bounds of the system memory. Use Soundo'le to create longer samples and then embed these samples into any application that supports OLE.

Other Windows utilities and applications include Mosaic, a tile game, Creative Talking Scheduler, a verbal reminder of appointments, and Creative Juke Box, a MIDI file player.

Sound Blaster 16 ASP Specifications	
Analog-Digital Input	5,000 - 44,100 Hz mono
	5,000 - 44,100 Hz stereo
Analog-Digital Output	5,000 - 44,100 Hz mono/stereo
Sample-Resolution	16-bit, 8-bit
Sample-Inputs	Mic mono
	Line-In stereo
	CD stereo
FM-Music	20 voice stereo
	20 voice OPL3 stereo
CMS-Music	No
CD-ROM Connector	Yes
Amplifier	4 Watt / 4 Ohm

1.1 Setup and Installation

As with any computer add-on, your Sound Blaster card must be installed before you can use it.

In this chapter, we'll explain how to configure and install your Sound Blaster card properly. We'll also discuss any special requirements for the individual versions.

Safety first

Precautions

Before unpacking your card, first touch a well-grounded metal object, such as a radiator pipe, to eliminate any static electricity. This protects your card from damage.

Most likely you've heard or seen a spark and felt a slight shock when you've touched a metal doorknob after walking across carpeting. The buildup of static electricity in your body is suddenly released in the same way as lightning during a thunderstorm.

Although as much as several thousand volts of electricity can be released, this isn't dangerous to you because the intensity of the current is extremely low. However, this electricity can destroy your card's electrical components.

So, although this precaution may not always be necessary, it's an easy way to avoid potential problems.

Unpacking your card

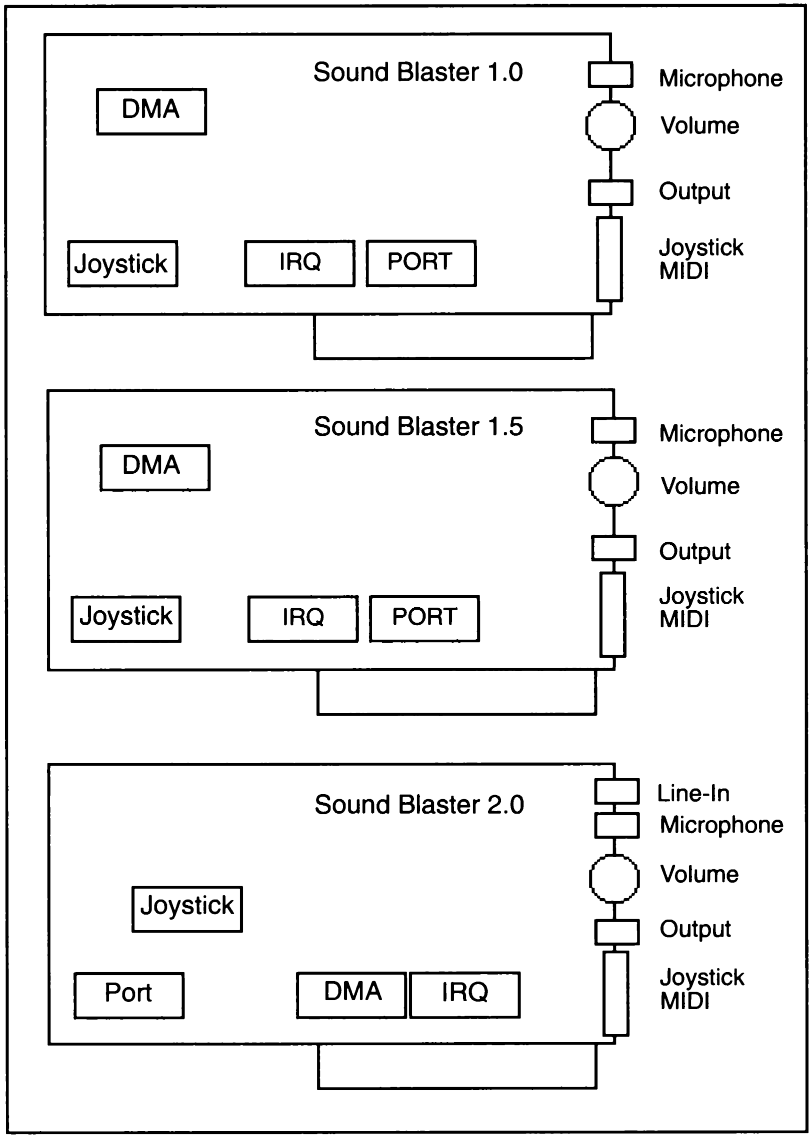
The first step

Now you can unpack your sound card. Some Sound Blaster Pro 1.0 packages contain a separate MIDI kit with the MIDI cable and sequencer software. We'll discuss these in detail later.

Another cable with a miniature (3.5 mm) plug and RCA phono plugs are also included in the Sound Blaster package. We'll also discuss this later. You should also find the software diskettes and then the actual card.

Remove the card from the package and place it in front of you on one of the foam packing mats.

The different Sound Blaster versions appear as follows (Sound Blaster 16 ASP is identical to Sound Blaster 2.0 in this illustration):



Different versions of the Sound Blaster card

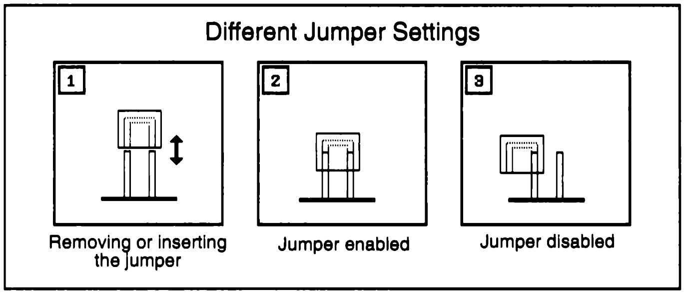
Jumper settings

As its name indicates, a jumper is a type of electrical connector that acts as a bridge on a circuit board.

Jumpers allow you to configure a board according to your own requirements, at the hardware level, by making, breaking, or

switching connections as needed at certain points on the board. This makes the installation more flexible and reduces the chance of conflict with other expansion cards.

The following figure shows how a jumper is removed and then reinserted:



Setting the Sound Blaster jumpers

The Sound Blaster card contains four important jumpers. Now we'll examine and test each jumper.

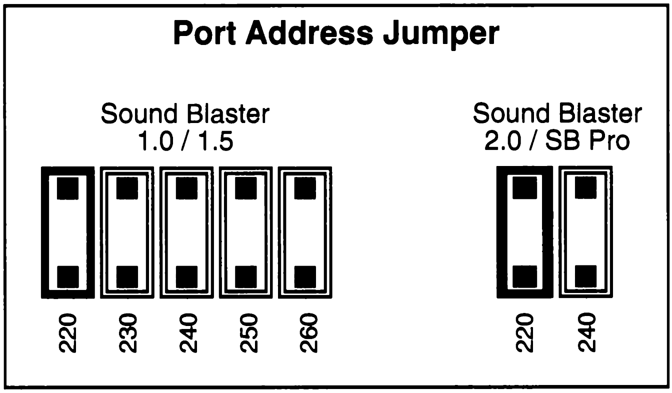
Port address jumper

Determine the memory addresses

This jumper allows you to determine the memory addresses through which your computer can reference the Sound Blaster card. The port address jumper is normally set for the base address 220 Hex. At this setting the Sound Blaster Pro, for example, would use the address range of 220 Hex to 233 Hex. The earlier cards use the address range of 220 Hex to 22E Hex. If another card in your computer occupies these same addresses, problems occur when both cards are being used simultaneously.

In this case, you can either reconfigure the other card, if possible, or set Sound Blaster to a different base address. Available port address jumper settings for Versions 1.0 and 1.5 are 210 Hex, 220 Hex, 230 Hex, 240 Hex, 250 Hex, and 260 Hex. For Version 2.0 and Sound Blaster Pro, you can select either 220 Hex or 240 Hex.

For a newly purchased card, the jumper is preset to 220 Hex. It's rare that another card would use this area. However, if this occurs, remove the jumper from the place marked 220 Hex and plug it in at the place marked for the desired port address.



Port address jumpers of Versions 1.0, 1.5, 2.0 and SB Pro

For more information, refer to Section 1.4.

Interrupt jumper

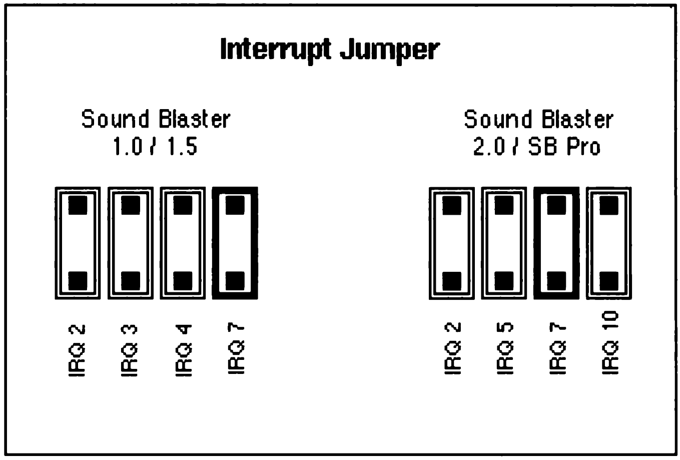
Similar to the port address jumper

The interrupt jumper is similar to the port address jumper. The interrupt that's least likely to result in conflicts is preset in a new card. This is interrupt 7, except in early models of Version 1.0, which used interrupt 3.

If you have an AT, you should use interrupt 5 or 7. Do not use interrupt 2, and use interrupt 10 only in special instances. Interrupt 3 is used mainly for a second serial interface.

Usually the default setting of interrupt 7 is suitable. However, problems can occur if you want to use Sound Blaster and print at the same time, since this interrupt is often used by the parallel printer connector.

If you want to change the interrupt selection, refer to the following illustration for the proper jumper setting.



Interrupt jumpers of Versions 1.0, 1.5, 2.0 and SB Pro

For more information, refer to Section 1.4.

DMA channel

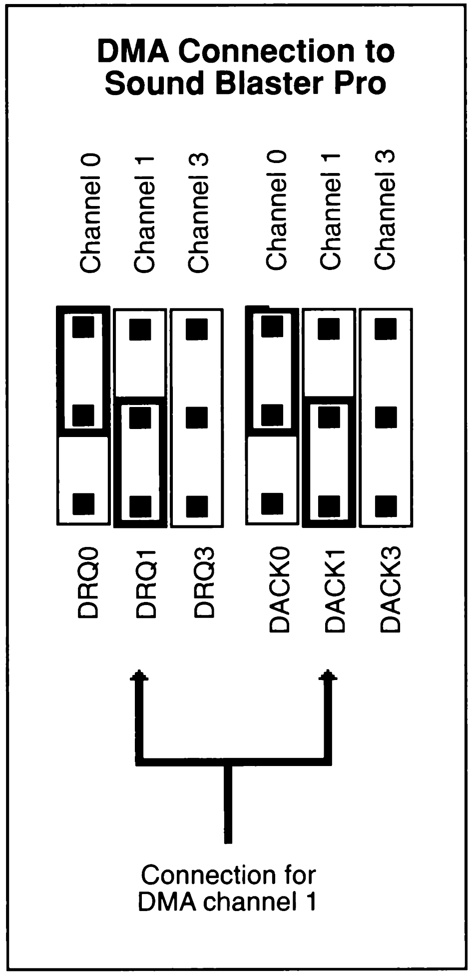
DMA jumper

DMA is an acronym for "Direct Memory Access." This refers to Sound Blaster's ability to access your computer's memory directly, without going through the CPU. This saves computing time and also allows sound processing to occur along with other tasks.

In Sound Blaster Versions 1.0, 1.5, and 2.0, the DMA jumper simply allows you to choose whether DMA is enabled. Removing the jumper disables the digital channel. In Sound Blaster Pro, however, you can choose between DMA channels 0, 1, and 3.

The default setting is Channel 1. If another card in your system is using this channel and doesn't allow DMA sharing (Sound Blaster Pro does support sharing), you should reconfigure the other card to use a different channel.

Changing the DMA channel selection on Sound Blaster Pro is slightly more complicated than other jumper settings, as the following figure shows:



DMA jumper on SB Pro

The jumper settings for DRQ and DACK must always match.

For more information, refer to Section 1.4.

Joystick

Joystick jumper

Most likely this will be the jumper that must be changed because of conflicts with other cards. Usually the conflict is caused by a multi-I/O card that contains not only parallel and serial interfaces but also an analog joystick.

I/O cards

The Sound Blaster card has a combined Joystick/MIDI connector. This connector will conflict with a multi-I/O card.

When this occurs you can either disable the joystick port on the multi-I/O card, if that's possible, or you can disable the joystick port on Sound Blaster.

If, instead of a multi-I/O card, you have a game I/O card that contains only a joystick connector, you can completely remove this card and connect your joystick to the Sound Blaster. Now you have an extra slot freed by the game card and Sound Blaster's joystick jumper doesn't have to be changed.

If you don't have a game I/O card or a multi-I/O card with a joystick port, then the Sound Blaster installation provides not only new sound capabilities, but also allows you to connect a joystick. For example, it's an excellent control medium for a flight simulator.

Obviously, there are many possibilities here. First determine whether you already have a joystick connector. If you have one, but can't or don't want to disable it, you must remove the joystick jumper from the Sound Blaster card. So if you want to use the joystick jumper at a later time, reinsert it in the card, as shown under "Jumper disabled" (see illustration on page 12).

Refer to Section 1.4 for more information.

PC beeper

Connecting the internal speakers

With Sound Blaster Pro you can also connect the signal of the internal speaker to your sound card.

However, to do this, you must be familiar with the speaker connections in your particular computer, and the suitable connection wires and sockets. Some experience with electronics is also helpful.

Don't try to do this yourself unless you understand the exact wiring requirements. A mistake could seriously damage the Sound Blaster card and your computer. If you need help, consult an experienced technician or your computer dealer.

If you do this yourself, first you must locate the internal speaker connections on your computer's motherboard. If possible, consult your computer's reference manual for a diagram of the motherboard. If this information isn't available, follow the cable from the speaker to the board. Then remove the speaker connection.

Now you must make two connections to the PC speaker pin on the Sound Blaster Pro. For details on this procedure, follow the instructions in your Sound Blaster manual.

We can't provide an exact description of the proper wiring technique because internal speaker connections vary among computers. However, don't switch the two connections or allow them to come in contact with the cable leading to the SB Pro. Therefore, be sure to use a jumper pin for the connection to the SB Pro card.

TIP

If you're not sure how to perform this process, or you have little experience in electronics, we suggest you avoid connecting your speaker to Sound Blaster output, and live with the standard PC beeps through the old speaker.

Installing the card

Ready for installation

Once you've made any necessary adjustments to the settings on the Sound Blaster card, you can begin the installation.

Disconnect, open, look inside

First switch off your computer and unplug it. Then open the computer case. If you have never done this before, check your computer manual for instructions because this procedure varies among computers.

To install Sound Blaster Version 1.0, 1.5, or 2.0, you need a free 8-bit or 16-bit slot. Sound Blaster Pro requires a 16-bit slot.

Unscrew the cover plate from the empty slot and set it aside.

Work carefully

Now position the card in the slot; the connector plate takes the place of the missing cover plate. Push down gently on the card to insert the contact pins into the socket. This is the most difficult part of the installation process. Once the card is properly seated in the slot, you can replace the screw.

If you drop the screw into the computer (a common occurrence), don't try to retrieve it with a magnetic screwdriver. Instead, touch the computer's power supply case (to discharge static from yourself), and reach in to pick up the screw if you can. If not, turn the computer upside down. The card won't fall out because it's very secure even without the screw, which is used more for grounding than for fastening.

When the Sound Blaster card is installed and all the jumpers have been properly set, you can put the cover back on your computer and plug it in.

Audio connections

Stereo system not required

Now that you've installed your Sound Blaster card, you'll obviously want to experiment with it. However, first it must be connected to a sound reproduction source. Several possibilities are available.

The Sound Blaster card has its own amplifier with 4 watts per channel output to 4 ohms impedance, and 2 watts per channel to 8 ohms impedance. So you don't actually need a stereo system to use Sound Blaster. We'll discuss some of the output alternatives here.

Connecting headphones

Headphones are the easiest connection

The easiest way to start using your Sound Blaster card is with a set of headphones. Simply plug the miniature (3.5 mm) headphone plug into the card's Audio-Output jack (see card diagram). You can still use your headset even if it has a different size jack (e.g., a standard plug). Simply obtain the appropriate adapter or actually solder a connection yourself.

However, you probably don't want headphones to be your only output medium. With headphones you can move only a limited distance from the computer. Also, you're cut off from the outside world. So you must connect speakers.

Connecting speakers

Speakers

When connecting speakers to your Sound Blaster card, you must consider the impedance of the speakers. You should use speakers with 4 ohms or 8 ohms impedance. The maximum output power of the built-in amplifier is then 4 watts or 2 watts, respectively. You can also use powered speakers (used for amplifying headphone output from personal stereos). The advantage of these speakers is that they allow individual volume control, so you don't have to adjust the volume control on the Sound Blaster card.

However, with Sound Blaster Pro, the output volume can be controlled through software.

Connecting a stereo

Stereo system

To hook up a stereo system, you'll need the connecting cable that was supplied with the card. This cable has a miniature (3.5 mm) plug at one end, and one red and one white RCA plug at the other. Plug the miniature plug into the Sound Blaster Audio Output jack, and hook the RCA plugs to the AUX input (or other LINE input) on your stereo.

If your stereo requires a different type of connector for the external sound source (e.g., a few boom box models use miniature [3.5 mm] jacks for AUX input), you'll need suitable adapters or a cable that's equipped with the required connectors.

If you've worked with electronic music devices before, and you have basic soldering skills, you should be able to rig up a cable that's customized for your needs. A small built-in switch could allow easy configuration changes in your system, by switching output between stereo and speakers, for example.

Once you've completed your connection to an output medium, turn the volume adjustment on your card to about midrange. Now you're ready to use the software.

Testing the installation

Was the installation successful?

Now you're ready to test your Sound Blaster card. You should make a backup copy of the installation software before you begin.

After you've copied the Sound Blaster diskettes, insert the first diskette in the drive. Start the TEST-SBC.EXE or TEST-SBP.EXE program, depending on whether you have Version 1.0, 1.5, 2.0, or SB Pro.

The test program ensures that your card is installed properly and that you've selected the proper jumper settings to avoid conflicts with other cards.

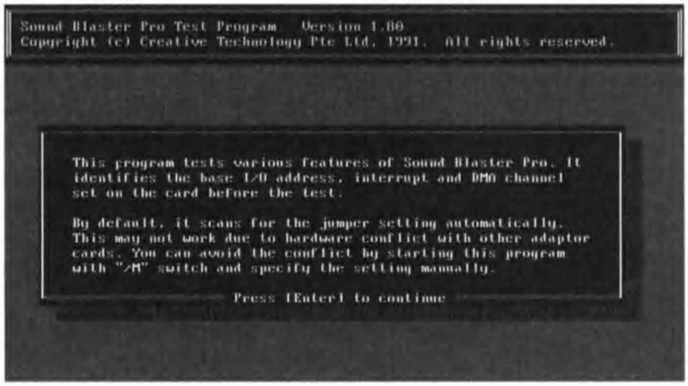
If a problem is detected, the test program helps determine its cause. In Section 1.4 we'll discuss possible problems in more detail.

Starting the test program

The text that appears on the screen for Sound Blaster installation programs varies, depending on the version you're using. Because of this, the illustrations in this section may not match the ones displayed with your Sound Blaster card.

So don't be surprised if things look different on your screen. Most of the important material is similar in all Sound Blaster installation programs.

The test program appears with this startup screen:

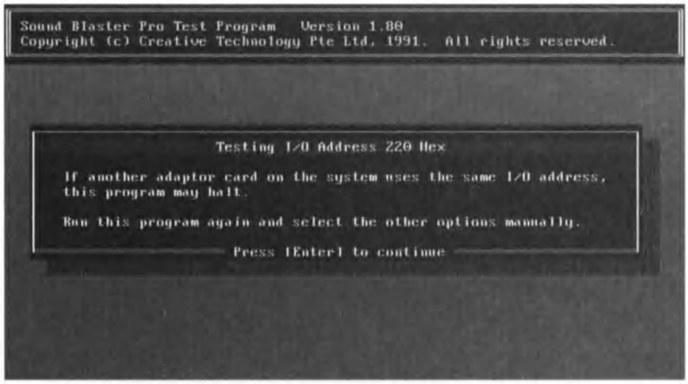


Test program startup screen

The program informs you that it will try to automatically recognize the jumper settings you've made. If the settings are incorrect and conflict with another card, your computer may "hang". In this case, start the test program with the "/M" switch. Now you can enter all the information manually, and the program won't try to determine your configuration by itself.

Testing the port address

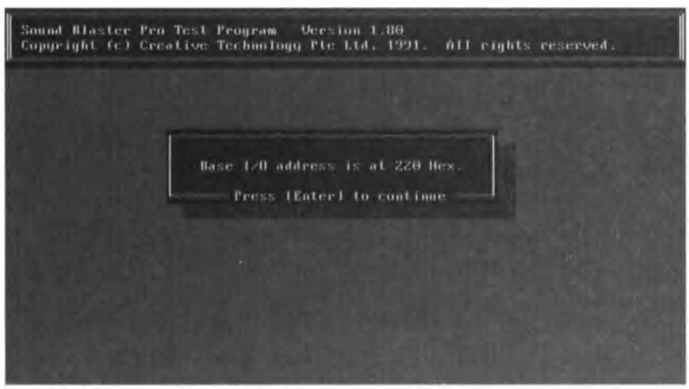
Before the program begins testing the port address, it displays another message, informing you that a conflict with another card could cause your system to hang.



The program tests the port address

Press Enter to have the program test the address.

If you've installed your card at the default value of 220 Hex and there is no conflict with any other cards, you'll see the following:



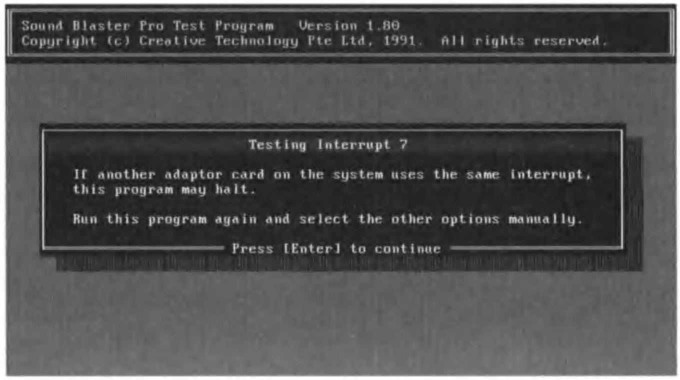
Sound Blaster address test successful

If the test program finds the Sound Blaster card at a different address, you'll see the value of this different port address.

Testing the interrupt

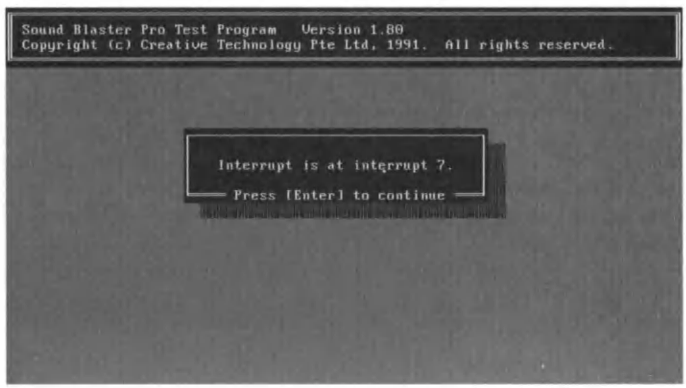
In the next step, the program tests the interrupt number. The default interrupt is Interrupt 7. This is the reason the program also checks this number first in the automatic test.

Again, if there is an interrupt conflict, the test program displays a warning message about hanging the system.



The program tests the interrupt

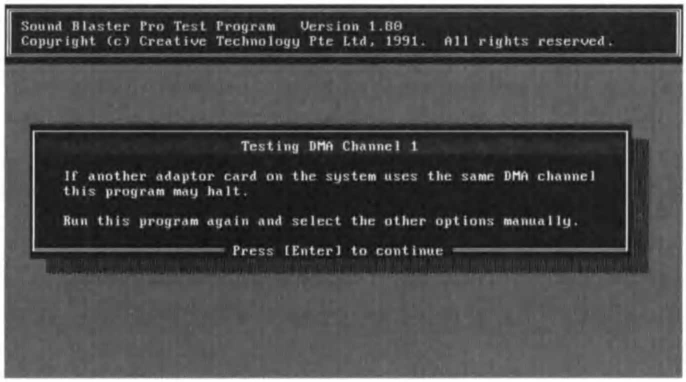
However, if the program finds the tested interrupt to be valid, you'll see the following message on the screen:



The tested interrupt is valid

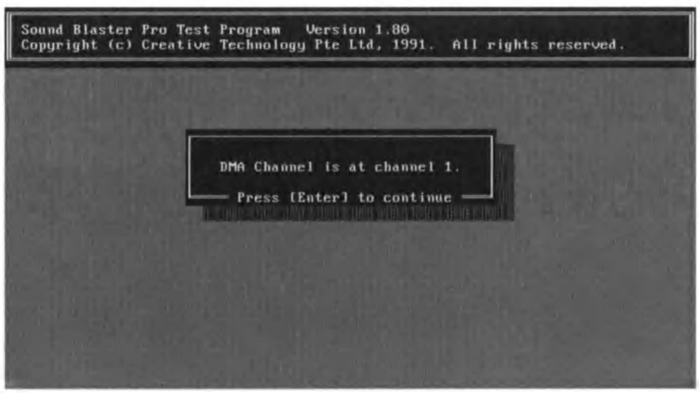
Testing the DMA channel

For the Sound Blaster Pro card, the program also tests the DMA channel. First you see the warning message.



The DMA channel is tested

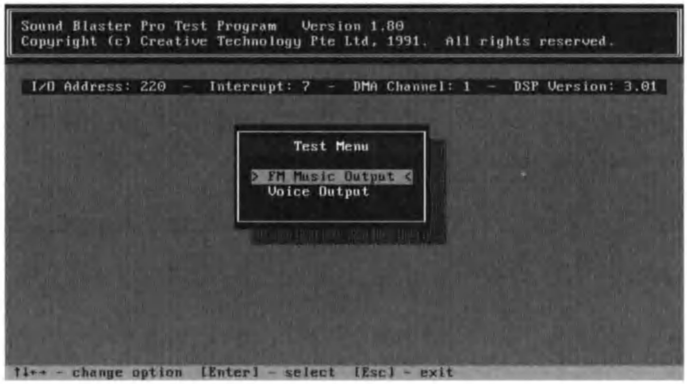
If the set DMA channel can be used without problems, the following message appears on the screen.



The DMA channel can be used

The first sounds

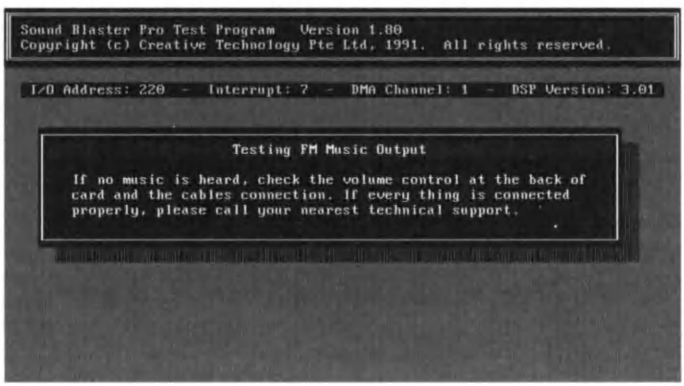
After the three tests are completed successfully, a small selection menu, from which you can choose to test either the FM voices or the digital channel, appears.



Now Sound Blaster can produce sounds

If you have a Sound Blaster Pro with OPL3 chips, you can also choose whether to test 2-operator FM voices or 4-operator FM voices.

Your card should play back FM music or a digital sound. The following are some tips on what you can do if you don't hear any sound.



The FM voices test

The volume control on the Sound Blaster card may be set to a setting that's too low. Ensure that the volume control is in a medium setting between Loud and Soft.

If you still don't hear any sound, test all the cable connections to the speakers, stereo, or headphones. If the cable connections are correct and the volume control is set correctly, but you still don't hear any sound, contact your dealer and ask him/her to check your card.

If all the tests are successful, then you've installed your Sound Blaster card correctly and you can begin using it.

1.2 Installing the Accompanying Software

Software installation

Now that the Sound Blaster card is successfully installed and the test program has provided a sample of the music you can play on your PC, you can install the Sound Blaster software.

In this section, we'll use Sound Blaster Pro as an example to explain the installation procedure. The process is almost completely automatic.

Step 1: Installation diskette

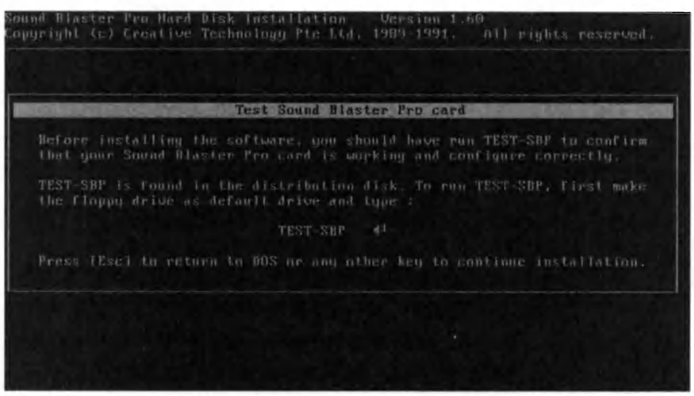
Insert the first of the supplied diskettes. You can identify this diskette by the number on the label. If the diskettes aren't numbered, insert the diskette containing the INST-HD.EXE program.

Depending on whether you inserted the diskette in drive A: or drive B:, now type "A:" or "B:" and press Enter.

Next start the program by entering the name of the program and the drive on which you want to install the software, for example, "INST-HD.EXE C:". The press Enter. The installation program starts.

Step 2: Installation program

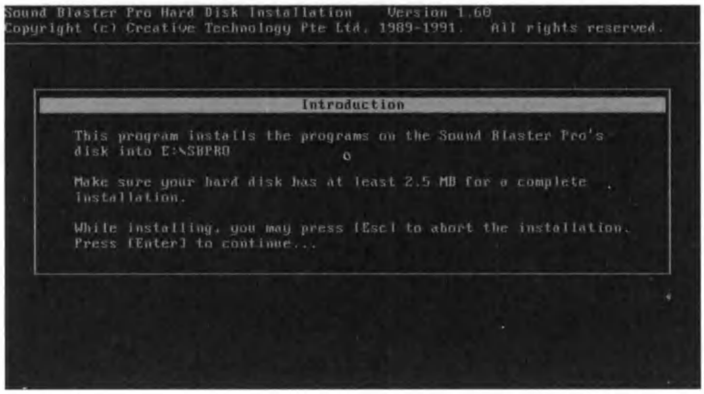
The first screen reminds you to start the test program before you begin installing the software.



The card test

Since you've already completed the test, press any key to continue or cancel by pressing Esc.

The next screen displays where the software is being installed and how much free disk space is needed.



Brief information screen before installation

If you don't have enough room to install, you can still cancel installation by pressing Esc. Press any other key to continue installation.

Next, the installation program automatically copies all the program files to your hard drive.

Then the program decompresses the files.

Depending on the type of diskettes (3.5-inch or 5.25-inch), occasionally you'll be prompted to insert the appropriate diskette.

Step 3: Sound Blaster environment

Environment variables

The next part of the installation program involves setting the Sound Blaster environment variables. The SET-ENV.EXE program automatically starts from within the installation program.

Setting the sound variables

This step has existed since Sound Blaster Version 2.0, since the newer versions of Sound Blaster software now support environment variables.

For more information on environment variables and instructions for changing the size of the environment area, refer to your DOS documentation.

The "SOUND" variable directs all applications, that prompt for the environment variables, to the Sound Blaster root directory.

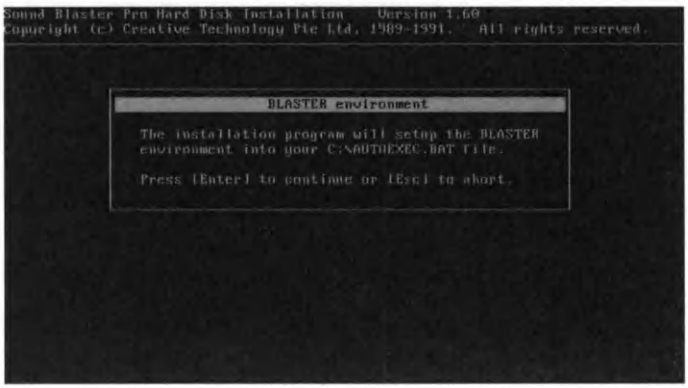
The setting is entered in the AUTOEXEC.BAT. Usually, the line looks as follows:

```
SET SOUND=C:\SBPRO
```

or

```
SET SOUND=C:\SB
```

Next, the program sets the BLASTER variable and enters it in your AUTOEXEC.BAT.



Blaster variable

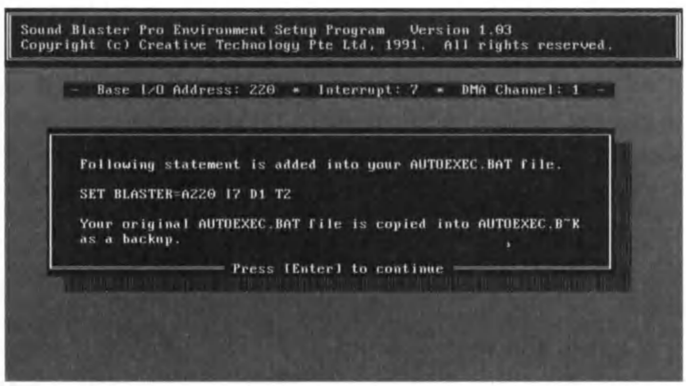
The BLASTER variable indicates the switches used to configure your Sound Blaster card. This is why the program must first check whether the card is correctly installed.

This should be familiar since it's the same procedure used by the Sound Blaster test program. The only difference is that for each test item, a menu appears. From this menu, you can choose the configuration switches yourself.

Determining the port address

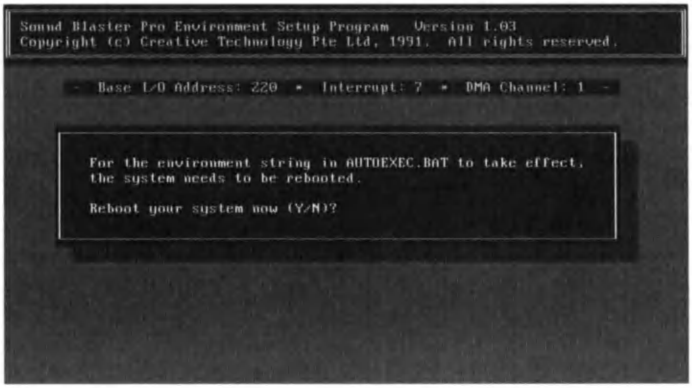
As we mentioned earlier, these same menus appear in the Sound Blaster Test program when you specify the "/M" switch. If you choose the Auto-Scan menu item each time, then the program will find the same configuration as the test program.

The SET-ENV.EXE program acknowledges the setting in the AUTOEXEC.BAT file.



The BLASTER variable is set

To make the environment variables active, you must restart the entire system. You can do this directly from the installation program.



The installation is complete

The software is now completely installed and the environment variables are set.

Installing the driver

Sound driver installation

If you've used the jumpers to change either the port address or the interrupt number of the Sound Blaster card, now you must provide new values for the sound driver.

This can also be accomplished by using a program. The INST-DRV.EXE program, which is located in the newly created directory (i.e., \SB or \SBPRO), helps you make changes to the CT-VOICE.DRV, ORGAN.DRV, and SBFMDRV.COM drivers.

If you call INST-DRV.EXE without command line parameters, it looks for the SBFMDRV.COM driver in the current directory. Then you can make changes to it as needed. To configure the other two drivers, the call must be made as follows:

```
INST-DRV \SBPRO\DRV
```

or

```
INST-DRV \SB\DRV
```

The program will display a list of drivers found in the directory. You can select CT-VOICE.DRV or ORGAN.DRV and apply the new information.

Getting started in the world of PC music

Music

Your Sound Blaster configuration is properly set. So you won't encounter port addresses, interrupts, and DMA channels any longer. However, be sure to write down the selected settings so you have them for future reference.

Some programs cannot automatically identify the configuration of the Sound Blaster card. This especially applies to many impressive graphics and sound demos, similar to those used on the Commodore Amiga. So you must provide these programs with information about your hardware configuration.

You must also enter these parameters if you want to use the Sound Blaster card under Windows.

Now you're ready to experience the world of PC music.

1.3 Accompanying Software

Sound Blaster software

We'll briefly discuss the software that's included with Sound Blaster and discuss a few specific software programs in detail.

We cannot discuss all the software in detail because the software differs depending on the version. Since the software market is changing rapidly, numerous programs will be developed in the future.

1.3.1 The software for the digital channel

The software consists of programs primarily intended for the digital channel of the Sound Blaster card. This includes small utility programs as well as large applications.

First we'll provide a quick introduction to the software that's shipped with Versions 1.0, 1.5, 2.0, as well as Sound Blaster Pro.

Don't be surprised if your Sound Blaster package doesn't contain one of these programs. Frequently, computer dealers selling the same version of Sound Blaster offer different software collections.

If you want a program that you don't have, check with friends and user groups. Public domain and shareware programs are also available through the Bulletin Board Systems (BBSes) free of charge.

However, remember that copying professional software is prohibited by law. Commercially distributed programs, such as Voice Editor, Voyetra Sequencer, and Santa Fe Media Manager can be purchased separately.

Intelligent Organ

Available in stereo

This program originally appeared in Game Blaster. However, it has been changed.

With Sound Blaster Version 1.5, which doesn't contain the factory-installed C/MS chips, this program was rewritten for FM voices. With Sound Blaster Pro, it's now available in a stereo version.

It has various capabilities and can be used with either a MIDI keyboard or with the keyboard on your PC.

The Intelligent Organ screen

The Intelligent Organ can record, save, and play back musical pieces. Unfortunately, the instrument sounds aren't very realistic and playing songs without a MIDI keyboard is awkward.

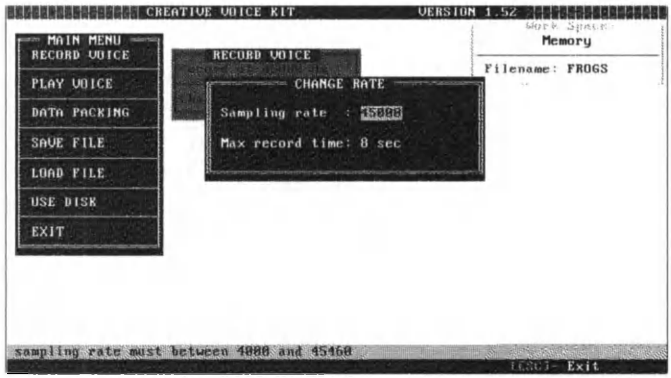
However, this program demonstrates the various possibilities of computer-supported music. For example, it offers features like bass accompaniment, arpeggio (notes sounded in succession), and artificial melody.

In general, Intelligent Organ is more of a game program than a professional music application.

VoxKit

VoxKit

The VoxKit program is used to record, save, and play sounds over the Sound Blaster card's digital sound input. It was included with Sound Blaster Versions 1.0, 1.5, and 2.0 but not with Sound Blaster Pro. Voice Editor, which was included with some Version 1.5s, has replaced this program and surpasses its capabilities.



VoxKit main menu and functions

The features mentioned are the basic elements of the VoxKit program. Also, the VoxKit program supports the Creative Voice File Format.

Talking Parrot

Good demonstration program

The Talking Parrot is a good demonstration program for the Sound Blaster card. Simply speak into a microphone, and the parrot on the screen mimics you, occasionally adding some comments.

It's very easy to run the program. After starting the program, a screen appears with a scale graphic indicating how strongly the Sound Blaster card picks up the noise of your microphone.

To prevent the parrot from interpreting the noise from your microphone as speech input, specify the level where you want the program to begin interpreting noise on the microphone as voice input. In other words, simply watch how high the lines on the displayed scale go when you aren't speaking into the microphone.

For example, if the lines go up to 140, enter 150 as your value. This guarantees that the parrot hears only your voice, not the background noise.

After entering the appropriate value, the friendly parrot appears on your screen and says hello. When you speak into the microphone, the parrot "responds". It makes comments or repeats what you said. The parrot always moves its beak when it talks.

You can cancel the program at any time by pressing Esc.

Customized parrot

You can also teach the parrot to say other phrases by replacing the various parrot files. For example, you could transform the talking parrot into a talking turtle or a talking refrigerator.

There are many possibilities. The following table shows which graphic files you must create to change the parrot.

Changing Parrot Graphics

Filename	Contents
PARROT.E0	Parrot with closed beak
PARROT.E1	Parrot with open beak
PARROT.E2	Parrot with wide open beak
PARROT.E3	Parrot with eyes and beak closed

You'll find these files in the Parrot directory. The files are PCX graphics in 16 colors, which are appropriate for an EGA or VGA card.

You can also place your own custom graphics under the same name. Remember to make a backup copy of the original parrot if you do this.

It's not as easy to place custom sounds in the sound files because all the sounds are in the PARROT.VCB file. To replace the phrases, you must create a total of 21 VOC files, from which you can produce a new PARROT.VCB file.

The following tables show which sounds you can use in the original version of the parrot program.

Random Parrot Greetings	
Filename	**Contents**
PVOC-A.VOC	Hello there
PVOC-B.VOC	Hi! How are you ?
PVOC-C.VOC	Good Day
PVOC-D.VOC	Welcome to the show
PVOC-E.VOC	I'm a talking Parrot
PVOC-F.VOC	Please talk to me
PVOC-G.VOC	Nice to see you
PVOC-H.VOC	Please say something
PVOC-I.VOC	Have a nice day
PVOC-J.VOC	Goodbye

NOTE

When you start the program, the parrot speaks messages A, E and D. At the end of the program, the parrot recites messages J and I.

Random Parrot Responses	
Filename	**Contents**
PVOC-K.VOC	Oh! You sound terrible
PVOC-L.VOC	Yak! You have bad breath
PVOC-M.VOC	What are you saying?
PVOC-N.VOC	What are you saying? (angry)
PVOC-O.VOC	Don't talk nonsense

NOTE

Sometimes the parrot refuses to repeat the text you enter. Instead, it provides one of these responses:

Parrot Responses To Keyboard Input	
Filename	**Contents**
PVOC-P.VOC	Ouch
PVOC-Q.VOC	Owwwww
PVOC-R.VOC	Don't touch me
PVOC-S.VOC	(Laugh 1)
PVOC-T.VOC	(Laugh 2)
PVOC-U.VOC	(Laugh 3)

The parrot responds to keyboard input with one of these messages.

You can create 21 different sample files (e.g., with the Voice Editor or VoxKit) and link them to another graphic. By doing this, you'll have an almost entirely different program.

File requirements

The sample files must meet a few requirements. The sample rate of the files must be at 10,000 Hz exactly, so when you play back the files, the recorded sounds don't undergo unintentional distortion. Also, the data must be in decompressed 8-bit format.

Single files cannot exceed the magic 64K limit. Otherwise, nothing will work. Taken together, all the files cannot be larger than the amount of available RAM when you load Parrot. The 21 original samples take up about 160K. If you stay within this framework you shouldn't have any trouble, and if you have enough free memory, you can even go beyond 160K.

When you have all 21 files together, copy them to the Parrot directory and start the MAKEPV.EXE program there. This program merges the 21 files into one new PARROT.VCB file.

After finishing these steps, you're ready to start Parrot and admire your work.

The Sound Blaster manual shows the files that contain graphics and sound information for this program. If you want to experiment, with a little manipulation you can switch these files to create a talking tortoise, refrigerator, or almost anything imaginable.

Voice Editor

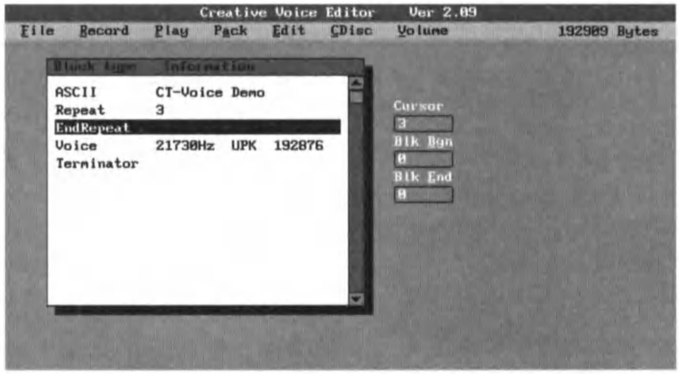
Very professional program

The most professional program among these software packages is Voice Editor. This program is available in two versions, one for Sound Blaster 1.0, 1.5, and 2.0, and a separate stereo version for Sound Blaster Pro.

This program provides all the capabilities needed to record, save, and play sound samples. Also, sample files can be extensively edited.

Voice Editor supports the complete Creative Voice File Format, including marker blocks, text and repeat loops. This distinguishes it from most of the editors in the public domain or shareware market. Since these editors support mainly the "raw 8-bit" format, they can handle only sample files consisting of only sound

information. The Creative Voice File Format is much more powerful and Voice Editor is an excellent tool for working with it.



The Voice Editor supports the Creative Voice File Format

Files that are too large to load into memory can be split into smaller portions that can be edited individually. Also, files can be played from or recorded to the hard disk directly. So the length of a recording is limited only by the capacity of your disk.

Voice Editor includes the most important sound editing functions. You can adjust the volume, add an echo, and fade the signal in or out. In the stereo version, you can even switch the signal from the left to the right speaker or vice versa.

Voice Editor effects

Cutting, moving, and copying segments of sound is easy. The only limitation is that the data cannot be packed. Once a data block is packed, it can no longer be edited.

Packing sound samples

With Sound Blaster you can support the packing of sample files at the hardware level. You can choose from among three compression ratios when packing files.

2:1 compression

A 2:1 compression ratio decreases the storage requirement for a sample by half. The maximum recording frequency for the sample file is 12 KHz. The amount of quality lost with 2:1 compression depends on the sample itself. However, this amount usually isn't noticeable.

When packing files, remember that you should always keep a copy of the original unpacked sample, since this cannot be recreated from the packed version. Also, packed data blocks cannot be edited.

3:1 compression

A 3:1 compression shrinks a file to one-third the original size. Quality starts to decrease noticeably at this ratio, and the maximum sample rate for 3:1 compression is about 13 KHz.

4:1 compression

A 4:1 compression produces a file one-fourth the original size. Quality decreases drastically with this ratio and the maximum sample frequency of the output file is about 11 KHz. The 3:1 and 4:1 compression ratios are suitable for packing noises like surf, gunfire, explosions, rockets, etc.

Silence packing

Another method of reducing sample files is called silence packing. With this method, you can specify two parameters. The first parameter is the threshold value, or maximum deviation allowed from a certain silence value for a sound to be considered silent. The second parameter is window size (i.e., the number of consecutive bytes that must be silent for packing to occur).

The program searches the sample file and removes data that corresponds to the selected parameters. A marker called a silence block replaces each gap. This marker is used only in the Creative Voice File Format.

With silence packing, it's possible that a file will split into so many blocks that it can no longer be edited. So, you should always keep an original of your file and pack it only after all other desired changes have been made.

Voice Editor is a very powerful program that's easy to use and fully compatible with the Creative Labs standards. This program also allows you to read files consisting of only sound data and convert them to the Creative Voice File Format.

With Voice Editor you can perform various sound experiments. You'll find many uses for this versatile program.

SBTalker

Speech synthesizer

The SBTalker program was included with some releases of Sound Blaster Version 1.5 and today is also likely to be included with SB Pro.

SBTalker is a very successful speech synthesizer that can read English text. It is a self-loading memory-resident driver. To save main memory, SBTalker will occupy EMS memory, if available. Once resident, it can be used by two programs, READ.EXE and SBAITSO.EXE.

READ.EXE can convert English text to speech in three ways. If you call the program without command line parameters, it reads text as you type. Exit this mode by pressing Ctrl + C.

It's also possible to include the text to be read at the DOS prompt when calling the program. For example, if you type:

```
READ Hello
```

SBTalker reads the word "Hello." To have SBTalker read an entire file (e.g., README.DOC), type:

```
READ.EXE <README.DOC /W
```

The "<" symbol indicates that the file's contents are input to READ.EXE; the "/W" parameter displays the text on the screen as it's read.

You should follow a few guidelines when creating the input text for READ.EXE. For example, remember that a string consisting entirely of uppercase letters will be read as a series of letters, instead of as a word. A numeric string, however, is read as a number instead of as a series of digits.

SET-ECHO.EXE is a companion program to the driver SBTalker. It adds an echo effect to the synthesized speech. After installing the driver, call SET-ECHO.EXE as follows:

```
SET-ECHO.EXE xxxx
```

The xxxx represents a number from 0 to 4000. This number is the approximate delay in 1/10,000ths seconds between speech and echo. You can create some interesting effects by varying this delay.

The REMOVE.EXE program removes the memory-resident driver SBTalker from memory and frees the space for other applications.

Although it's fun to use SBTalker, it's also capable of high-quality speech. So you can also use SBTalker for serious applications.

Dr. SBaitso

SBTalker application

This program is an example of an SBTalker application. You can determine for yourself how serious an application it is. However, talking with Dr. SBaitso should be an enjoyable experience.

Dr. SBaitso is a computer psychotherapist who wants to help you solve your problems. This program is similar to some programs from the early 1980s or before. You may be familiar with one program called ELIZA, which simulated a psychologist.

Sound Blaster includes the SBAITSO.EXE file, while SB Pro has the SBAITSO2.EXE file. Make sure SBTalk is installed, then run the appropriate file.

Dr. SBaitso takes parts of sentences that you type as input, changes them around to form questions or comments, and then tries to create the illusion that you're exchanging ideas with an intelligent partner. Unlike its predecessors, Dr. SBaitso is able to vocalize his responses in addition to printing them on the screen.

Dr. SBaitso in stereo

If you have Sound Blaster Pro, you can activate Dr. SBaitso with the "/S" parameter. This allows you to hear the complete dialog in stereo. Dr. SBaitso's voice is played over one channel and your own sentences are spoken in a different voice over the other.

You can ask for "HELP" to display a screen containing information on how to conduct your conversation. Try experimenting with this program. Dr. SBaitso always has some surprises. He can even perform a few simple calculations for you.

For example, try telling him to "CALC 4 * 5". He also has some favorite topics. For example, try discussing money with him.

DOS Tools

Utility programs

Now we'll discuss some utility programs that can be used at the DOS level with the help of command line parameters. These programs are VOC-HDR.EXE, JOINTVOC.EXE, VREC.EXE, VPLAY.EXE, WAV2VOC.EXE, and VOC2WAV.EXE. You'll find these programs under the Voice Editor or VoxKit directory if you have one of these programs.

The functions of the first four utilities are included in Voice Editor, so you may not need these.

However, the VPLAY.EXE program is quite useful.

VPLAY

VPLAY allows you to play a VOC (voice) file of any size. The file is played through a buffer directly from a diskette or hard disk.

For example, you can use this program to have your computer greet you when it's switched on. To do this, use a digitized "Welcome!" or "Hello!" at the end of the AUTOEXEC.BAT file.

VOC-HDR

Occasionally you may have to use the VOC-HDR.EXE program. This is usually necessary when you try to load a raw 8-bit sound file, which exceeds a certain size, into Voice Editor.

Until the file is converted to Creative Voice File Format (i.e., until it has a CT Voice header), Voice Editor cannot split the file into loadable sections. Instead, it quits with the message informing you that the file is too large.

At the same time, the program can't convert the file without first loading it. With smaller files, which can be loaded successfully, Voice Editor adds the header automatically after asking you for the sample rate.

If an Error 5312 occurs, VOC-HDR can help. Copy the program to the directory where the sound file is located, if this directory isn't included in your DOS search path. Then type the following command:

```
VOC-HDR.EXE SOUND.DAT SOUND.VOC
```

Now press Enter. A menu asks whether the file is an 8-bit, 4-bit, 2.6-bit, 2-bit, or stereo file.

This program can process files that have been packed. A 4-bit file results from a compression ratio of 2:1, 2.6-bit results from 3:1 compression, and 2-bit from 4:1 compression.

After making a selection, you're asked for the sample rate of the file. If you didn't record the sound yourself and don't know the actual rate, simply guess. You can correct it later with Voice Editor. VOC-HDR then converts SOUND.DAT to the desired VOC file, which you can edit in Voice Editor.

The VOC2WAV and WAV2VOC programs are treated separately in reference to Microsoft Windows and Multimedia. They first appeared with Sound Blaster Pro.

There are also other tools that accompany many of the Sound Blaster versions. However, since the capabilities of Voice Editor surpass most of these tools, we won't discuss them here..

1.3.2 Software for the FM voices

Unfortunately, there isn't much support for programming FM voices. However, this doesn't apply to only the supplied software. FM voices haven't been favorably received in the world of public domain and shareware either, unlike the programs written for digital channel. The only program actually designed for FM voices is the Intelligent Organ.

1.3.3 Multimedia software

The software programs discussed in this section are considered multimedia programs. Most of these programs are included with Sound Blaster Pro. As long as they aren't designed only for use with the special capabilities of the Sound Blaster Pro, you can also use these programs on the other Sound Blaster cards.

Multi Media Player

Create your own presentations

With MMPlay you can combine impressive graphics, music, and digital sound to create your own presentations.

The basis of the presentation is a script file that describes the type and sequence of music and graphic animation to be played. MMPlay supports the CMF format of Creative Labs for music, the VOC format of Creative Labs for digital effects, and, for graphic animation, the FLI format of Autodesk, as is utilized by the program Autodesk Animator.

Even if you don't own Autodesk Animator, you can still experiment with MMPlay by obtaining some of the public domain FLI files and perhaps setting them to music. Obviously, it's more challenging to create everything yourself, from music, to graphics, to special effects.

The script language of MMPlay

The MMPlay script language consists of 16 commands that are used to control the execution of graphics and sound.

By conforming to the Creative Voice File Format, graphic animation, music and digital sound effects can be synchronized with one another. Synchronization is accomplished with the use of marker blocks within sample files and music segments. Voice Editor can add such markers to your sample files. CMF music files can also contain markers. However, currently there is no suitable program for editing CMF files.

Combining music, digital sound effects, and animation is called multimedia. With MMPlay, you can create impressive multimedia demonstrations.

The following is a brief introduction to the commands of the script language. Simply enter the desired commands in an ASCII text file that you can create with any text editor or word processor.

Each script command starts with a period for its first character, for example ".APLAY" or ".PAUSE".

.APLAY

Autodesk animation files

This command loads an Autodesk animation file and plays it back. For example, the command line ".APLAY intro.fli" loads the INTRO.FLI file and plays it back. You can also omit the FLI extension from the filename.

The file will continuously start over at the beginning until a synchronization condition occurs. We'll discuss synchronization conditions in more detail when we reach the .SYNC script command.

However, you can also interrupt the playback by pressing a key.

.APLAY1

This command is basically the same as .APLAY. However, the specified animation file plays back only once and then it stops.

.DELAY

Inserts a time break

The .DELAY command inserts a break in a presentation before the next command is executed. Specify the time in hundredth seconds. For example, enter ".DELAY 200" to produce a 2 second pause.

.PAUSE

Use the .PAUSE command to stop a running presentation until a key is pressed.

.PLAY

Outputting CMF files

Use the .PLAY command to start outputting a CMF file. For example, the ".PLAY funny.cmf" command line loads and starts a file called "FUNNY.CMF". You also have the option of omitting the CMF extension.

Before starting the presentation, make sure you load the memory resident driver "SBFMDRV.COM". Otherwise, you won't be able to hear any FM music.

.PLAYCD

Adding music from audio CDs

You can also add music from an audio CD to your presentation. Use the .PLAYCD command for this purpose.

You must specify at least the number of the title, for example ".PLAYCD 3". This command line plays back the third title of the current audio CD.

However, you can be even more precise by specifying a start time and a duration. The command line ".PLAYCD 3 152 42" plays back the third title, beginning at the 152nd second, for exactly 42 seconds.

.REPEAT .END

Repeating commands

The .REPEAT command specifies how often the commands in the following lines should be executed. This command affects all the lines between ".REPEAT" and ".END".

Specify the desired number after ".REPEAT", for example:

```
.REPEAT 5
.APLAY1 bounce.fli
.END
```

These three lines cause the BOUNCE.FLI file to play back exactly three times.

.SCREEN

Changing modes

With the help of the .SCREEN command, you can choose to have the screen remain in graphics mode or switch back to text mode at the end of a presentation. Enter ".SCREEN 0" for graphics mode or ".SCREEN 1" for text mode.

.SYNC

The Creative Labs formats for digital channel and FM voices both provide the option of placing synchronization information within the musical data.

With the .SYNC command, users can use this information to control animations. For example, by entering the script line ".SYNC V4711", you can specify that the next command, .APLAY, cancels an animation upon receiving the condition "4711" from a played back VOC file.

You could also use the command .SYNC F0815 to cancel the following .APLAY command when a CMF file running in the background sends a status of 0815.

The letter "V" refers to status information from a VOC file while the letter "F" represents information from a CMF file.

.STOP

Ending output

The .STOP command ends the output of a sound file. The command line ".STOP V" cancels the playback of a VOC file, while ".STOP F" stops a CMF file.

.STOPCD

There is a separate command for stopping an audio CD. Enter ".STOPCD" to stop an audio CD.

.VOLUME

Setting mixer chips

Use the .VOLUME command to set the mixer chips of the Sound Blaster Pro.

You can supply each component with volume information. Use the values "CD", "LINE", "MASTER", "MUSIC" and "VOICE" as

switches. The values for the left and right channels can be between 0 and 255.

For example, enter ".VOLUME MUSIC 150 250" to set the volume for the FM voices at 150 for the left channel and 250 for the right channel.

.VOUT

Playing VOC files

Use .VOUT to load and play back a VOC file. As an example, the command line ".VOUT welcome.voc" loads the "WELCOME.VOC" file and then plays it back.

.WAIT

Wait signal

.WAIT is comparable to the .SYNC command. It interrupts a running presentation until a certain status occurs.

For example, enter ".WAIT V4711" to pause a presentation until a played back VOC file sets a status of 4711. Enter ".WAIT F0815" to interrupt a presentation until a CMF file sets a status of 0815.

Santa Fe Media Manager

With the Santa Fe Media Manager, currently only available with certain European editions of Sound Blaster Pro, you get a professional multimedia database.

Data, graphics, animation, sound

Santa Fe Media Manager lets you create database files and provide them with graphics, animation, sound, and text information. The only requirements are a VGA card with at least 512K of memory and a resolution mode of 640x480 pixels with 256 colors, and at least 320K of free extended memory.

Also, the program, including a demo database, takes almost 5 Meg of hard disk space. However, this program provides everything you need to enter the exciting world of multimedia databases.

The database format

If you work with Borland's Paradox database, you can use your data in Santa Fe Media Manager because it supports the Borland format. You can also create new databases with the Santa Fe Database Creator.

If you use a different database and want to convert your data for Santa Fe Media Manager, you can import it from an ASCII file, in which a particular separation indicator has been used to mark the

individual data records. Your database system must have an ASCII export function.

Graphic formats

Since the PCX and GIF graphic formats are supported, you have access to the largest collection of PC clip art available. An utility program can convert your graphics if they're not one of the required formats. The uses for graphics in multimedia are almost endless.

For example, suppose that you collect stamps. With a color scanner for input, you can include a picture of each stamp in your collection database.

Animation format

You can incorporate animation in Santa Fe Media Manager as in MMPlay, again by using the FLI format of Autodesk. When developing your own creations, Autodesk Animator is very helpful because you can develop the animation according to your needs.

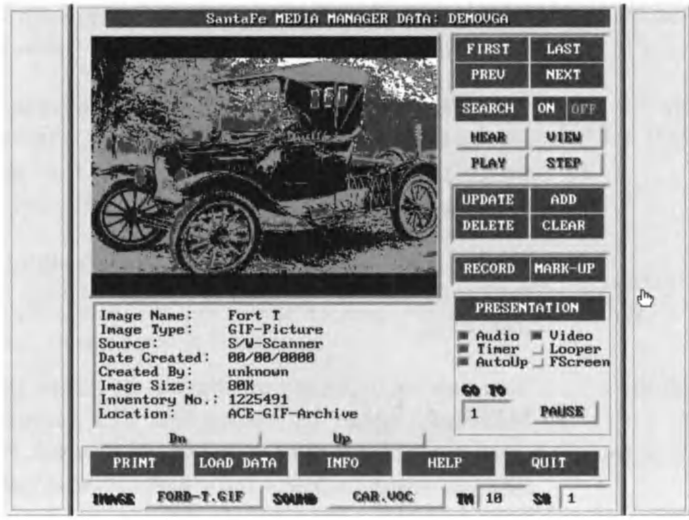
However, animation files from other sources also provide numerous possibilities. What could be better than having your own video film manager, selecting the "Star Wars I" data record, and watching an X-Wing fighter race across the screen? You can also use public domain animation files.

Sample format

The sample format supported is the Creative Voice File Format. You can also use Apple Macintosh format samples by converting the samples using the Voice Editor utility.

As in any good database system, you can search for particular data records, then modify or print them, etc.

The Santa Fe Media Manager package has many utilities of its own. Even if you've never worked with software for editing graphics or recording sound samples, you'll quickly be using the Media Manager for various tasks, such as building a new database or creating a finished presentation.



The Santa Fe database management screen

Power, versatility, and simplicity make the Santa Fe Media Manager a valuable package for both beginners and professionals.

SBSIM program

For programmers

The SBSIM program is a memory-resident driver that manages the various individual sound drivers.

SBSIM loads the drivers CT-VOICE.DRV, CTVDSK.DRV, and AUXDRV.DRV independently as needed into memory. You provide a configuration file that indicates the necessary drivers and the parameters for calling them.

The only driver SBSIM cannot load is SBFMDRV.COM, because this driver itself must reside in memory. So this means that you must load this driver before starting SBSIM.

Multiple programs have external access to the driver manager SBSIM. An application program that utilizes its services doesn't need to load its own version of CT-VOICE.DRV, for example, when it calls this driver. SBSIM ensures that CT-VOICE.DRV is available.

Also, SBSIM supports the use of extended memory with the definition of memory handles. Your memory manager must be compatible with the functions of HIMEM.SYS.

SBSIM has supplementary utilities for playing sound samples and CMF music segments. A MIDI file player is also available. The parameters of these utilities are very comprehensive and provide everything necessary for VOC and CMF file output.

Another utility program is SOUNDFX. This program can be used to produce such effects as fading (i.e., bringing a sound in or out) and panning (i.e., changing stereo channels) for all the sound input or output sources of Sound Blaster Pro. You can create numerous effects by using this utility.

Special Sound Blaster Pro programs

New utilities

In the previous sections we discussed the software that's included with Sound Blaster Pro.

However, Sound Blaster Pro is different in many ways from earlier versions of the Sound Blaster card. Because of this, new utilities have also been written to take advantage of SB Pro's special features.

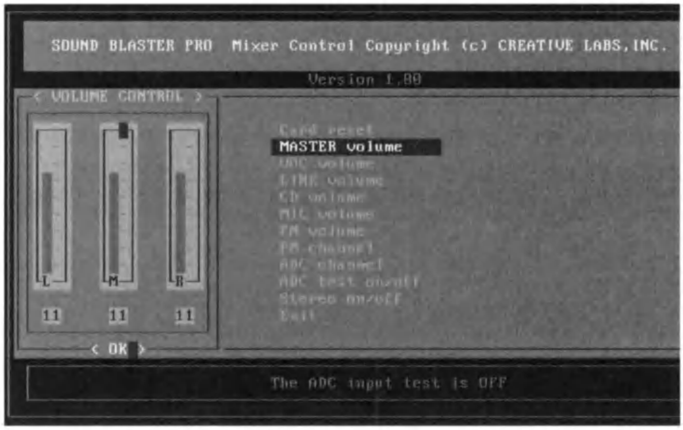
SBP-MIX program

Sound Blaster Pro provides various signal input and output selections, which can be controlled individually by the software. On-screen control takes place via the sound-mixing program SBP-MIX.

When called, this program installs itself as memory-resident and occupies about 78,000 bytes.

Normally the hotkey combination Alt + 1 is used to activate SBP-MIX. Under certain circumstances, this combination can be changed. To remove the program from memory, call it with the command line parameter "/U" (for "Unload").

Using SBP-MIX, you can adjust the volume of each of the components of the sound card (master volume, FM voices, CD player, etc.).



Main menu of SBP-MIX

As a memory-resident program, SBP-MIX can also be called by other applications. It even functions in a graphics mode. In this mode, instead of the menu that's displayed in text mode, only the last line of the screen displays the mixing function commands. Use the ↑ and ↓ arrow keys to select a function, and the ← and → arrow keys to adjust its setting. To leave the program, select Exit.

Under certain circumstances, using the hotkey to activate SBP-MIX from within an application can cause problems. In these instances, you may have to select your volume settings with a different utility (SBP-SET.EXE) before starting the application.

Although this procedure is less convenient, you can gain 78,000 bytes of valuable main memory by not loading SBP-MIX.

SBP-SET program

Specifies settings for the Sound Blaster Pro card

SBP-SET.EXE allows you to specify all the settings for the Sound Blaster Pro card using command line parameters.

You'll probably need to use this program eventually, perhaps even as part of your AUTOEXEC.BAT file (where it can be used to set volumes). So we'll discuss each parameter in detail:

Parameter /Q

Quiet screen mode. All screen output is suppressed except for error messages.

Parameter /R

Reset. All Sound Blaster Pro parameters are reset to their original default settings.

Parameter /ADCF:xx

Analog-digital conversion filter. Specifies the filter for recording the analog signal; xx is either LOW or HIGH. The default setting is LOW.

Parameter /ADCS:xx

Analog-digital conversion source. Specifies the signal source to be recorded; xx can be MIC for microphone, CD for CD player, or LINE for Line-In.

Parameter /ANFI:xx

Analog noise filter. Specifies whether to use a noise filter for digitizing (recording); xx can be ON (use filter) or OFF (don't use filter). The default setting is ON.

Parameter /DNFI:xx

Digital noise filter. Specifies whether to use a noise filter for playing digital sound; xx can be ON (use filter) or OFF (don't use filter). The default setting is ON.

Parameter /M:l,r

Master volume. Specifies master output volume; l and r are left and right channel settings, with volume values from 0 to 15. The default setting is 9 for both left and right.

Parameter /VOC:l,r

Voice volume. Specifies volume for digital audio channel. The default setting is 9 for both left and right.

Parameter /FM:l,r

FM volume. Specifies volume for the 22 internal FM voices. The default setting is 9 for both left and right channels.

Parameter /LINE:l,r

Line-In volume. Specifies volume for the Line-In signal. The default setting is 0 for both left and right channels.

Parameter /CD:xx

CD volume. Specifies volume for the CD player; xx is a value from 0 to 15. The default setting is 0. There is no separation of left and right.

Parameter /X:xx

Microphone volume. Specifies volume for the microphone; xx is a value from 0 to 7. The default setting is 0.

1.4 Solving Hardware Problems

Hardware problems

Now we'll discuss some the hardware problems that can occur when you're installing your Sound Blaster card. We'll take a more detailed look at port addresses, interrupts, DMA channels, and joystick jumpers.

1.4.1 Port addresses

Problems

If your system has an I/O address conflict between Sound Blaster and another card, the test program TEST-SBC or TEST-SBP will crash when it tries to access this address. When this occurs, you must restart your system with a warm boot.

Solution

After determining which card is responsible for the conflict, you might be able to assign it to a different address and avoid having to reconfigure the Sound Blaster card. Otherwise, simply switch the Sound Blaster's port address jumper.

Version 2.0 and Sound Blaster Pro contain only the choices 220 Hex and 240 Hex. This should be enough to resolve the conflict. Even on Versions 1.0 and 1.5, the recommended second choice, after 220 Hex, is 240 Hex. A conflict rarely occurs with these addresses.

1.4.2 Interrupts

Problems

If the test program TEST-SBC or TEST-SBP crashes when checking Sound Blaster's interrupt line, your system has an interrupt conflict. A warm boot is needed to restart. Reconfiguring the Sound Blaster card is probably the easiest way to resolve an interrupt conflict.

There are several interrupts which are used for various tasks. The ones you'll be using are those numbered 0 through 10. The following table shows how these interrupts are normally used:

Interrupt	Function
0	System timer
1	Keyboard
2	Not used
3	COM 2 (or not used, if no COM 2)
4	COM 1
5	Not used
6	Diskette controller
7	LPT1
8	Clock/calendar
9	Not used
10	Not used

With Sound Blaster Versions 1.0, 1.5, and 2.0, you can choose an interrupt setting of 2, 3, 5, or 7. Interrupt 2, 5, 7, or 10 can be selected for Sound Blaster Pro.

On some systems, Interrupt 7 is reserved by the printer port LPT1:, although it isn't actually used. In this case, your computer crashes if you try to play sounds with Sound Blaster while printing. You can solve this problem by changing Sound Blaster's interrupt. The next best choice is interrupt 5.

Interrupt 7 and OS/2

You should also make this change to your Sound Blaster card if you're using Sound Blaster under OS/2. Here interrupt 7 is used by LPT1: for printing. If the Sound Blaster card is configured to use this interrupt, you'll probably encounter problems when printing.

1.4.3 DMA channels

DMA channel

DMA channel conflicts are also detected by the test program TEST-SBC or TEST-SBP.

Only Sound Blaster Pro enables you to change DMA channels. With Versions 1.0, 1.5, and 2.0, you can only choose whether to enable DMA access. Disabling DMA access, however, means that the digital audio channel cannot be used. If a conflict occurs with one of the earlier cards, determine its cause by finding out which other card is using DMA channel 1. Usually this is a scanner or a network card.

If this card is able to share a DMA channel, the conflict isn't a problem because Sound Blaster is also capable of sharing. Otherwise, it may be possible to reconfigure the other card. You

must have at least a 286 AT or compatible computer to do this, because an XT doesn't have another free DMA channel.

If the other card cannot be reconfigured, you must avoid using both cards at the same time.

On an AT system with Sound Blaster Pro, you have much more flexibility. However, you should try all the other options before changing the Sound Blaster setting from the default of DMA channel 1. If no other alternative is possible, change the jumper setting to channel 0 or channel 3.

1.4.4 Joystick jumper

Two types of joystick problems can occur in your system.

Inoperable joystick

With the first problem, your joystick doesn't work regardless of which program tries to use it. If the joystick isn't defective and is properly connected to the joystick port, your system probably has another joystick connector besides the Sound Blaster's.

You can either locate the other card and disable its joystick connector, if this is possible, or disable Sound Blaster's joystick connector by removing the jumper, as described earlier. Then your joystick should operate properly.

Position calculation

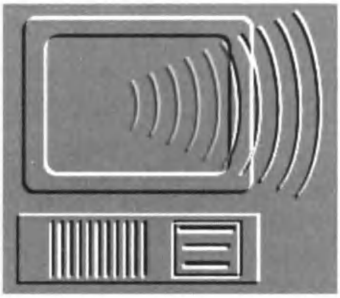
Another problem occurs with the calculation of the position of an IBM joystick. Although this isn't related to the Sound Blaster card itself, it's a common problem that occurs with joysticks.

If your joystick movements produce an erratic response, the program is probably using CPU timing to calculate the joystick position. Miscalculations can occur on computers that are too fast for the program.

Unfortunately, the only way to solve this problem is to slow down your computer. However, you probably don't want to do this.

One alternative is to use the keyboard instead of the joystick. Again, remember that the Sound Blaster connector isn't causing this problem. The same problem would occur with any joystick connector card.

This problem usually occurs with older computer games. New joystick algorithms, which can handle the high speeds of modern CPUs, have since been developed.



Chapter 2

DOS Software

In this chapter we'll discuss the software that enables you to fully use Sound Blaster's capabilities, especially its digital audio channel feature. Remember, these are just a few of the software programs that are available. Obviously we cannot discuss all the software.

The programs presented in this chapter will run on your computer even without Windows. We'll discuss the Sound Blaster programs that run under Windows in Chapter 3.

2.1 Application Programs

Application programs

Only a limited number of application programs provide sound support under DOS. These programs are usually music programs or presentation software.

Voyetra Sequencer

One of the best music programs is the MIDI sequencer by Voyetra. Some packages of Sound Blaster Pro 1.0 include a version of Voyetra Sequencer that's specifically tailored for this card. In Chapter 4 we'll discuss this version in detail.

Other than this, most application software don't use an available Sound Blaster card. Unfortunately, this segment of PC programming isn't very advanced yet.

2.2 Games

Sound Blaster is used with many computer games. Great sound is an important part of a game's appeal. Because of this, PC sound is more advanced in games than in application software.

2.2.1 Types of games

Simulations, strategy, sports games

Simulation games can be identified by their exact detail and close resemblance to reality. Many flight simulators and other simulation games already offer realistic graphics, but their sound effects could be improved.

Most simulation games feature only basic sounds, such as engine noise, screeching tires, and explosions. Of course, these sounds are usually sufficient for the situations these games simulate.

Therefore, most of these games don't extensively use sound cards. However, Falcon 3.0 is an example of a flight simulator that effectively uses sound capabilities. In this game, digital graphics and animation are combined with good sound to achieve impressive results. The game is especially exciting when it's played over a network by several players.

Usually sound is less important in thought, strategy, and sport games than in other types of games (e.g., action games). The games presented in this section make good use of Sound Blaster's capabilities, even if the sound isn't as spectacular as some action game effects.

Indianapolis 500

Indy 500 was one of the first games to not only offer good title music over the FM voices, but also continue to play sound effects while the games is being played.

This game places you in the 500-mile auto race at Indianapolis. You begin by getting your car ready, making decisions on the transmission gearing, and the profile and tilt of the tires. After going through trials to find the optimal settings, you and your fine-tuned machine are ready to start.

Instant replays

During the race the action is captured for instant replays, and because of solid-filled vector graphics you can even view an event, for example an accident, from different angles.

If you enjoy working with cars and don't find it too boring to drive 500 miles in left-hand curves, Indy 500 may interest you.

Links

Golf simulation

Although Links is very similar to the game Leaderboard, it's a much better game. Links is more than a simple golf game; it's a golf simulation.

Since you can adjust several parameters, you'll feel like you're really standing on the golf course. You control the player's foot placement and the angle of the club against the ball. Digitized VGA graphics provide excellent animation of the golfer on the screen.

First-rate vector graphics

The landscape is created with first-rate vector graphics. A fairly fast computer helps handle the high-precision detail. Besides the basic course, new golf courses for Links are also available on diskette.

The sound is also excellent. Birds chirp and spectators applaud your successes or murmur as the ball barely misses the cup. Also, your talkative caddy frequently comments on your shots.

The digital sound effects are played over the Sound Blaster's digital audio channel. However, even without a sound card, Links delivers your money's worth with its RealSound system, which was a feature in Leaderboard. Links is one of the best golfing programs currently available for the PC.

A version that fully uses the Super VGA functions, called LINK386PRO, is now available.

Winter Challenge

Compete in several winter sports

In Winter Challenge from Accolade, you compete for medals in various winter sports. These sports include giant slalom, speed-skating, ski-jumping, tobogganing, and bobsledding.

You don't have to finish a competition in one session. It's possible to save your standings and resume the competition at another time.

The music and fanfare at the beginning of each event is sent over the FM voices of the sound card. The digital audio channel delivers the whoosh of your skis on powder and the roar of your sled as it passes under a bridge.

The program uses both modes of Sound Blaster to create sound, with emphasis on the FM side. If you enjoy winter sports and the Winter Olympics, you should check out Winter Challenge.

Shanghai II

In the early days of the Commodore Amiga, a game called Shanghai was produced by the firm Activision. This game was an adaptation of an old Chinese game that's played by removing

pairs of stones from a pile until no stones remain. The stones are stacked in a certain order and can be removed only in matching pairs. It may remind you of a similar game called Memory.

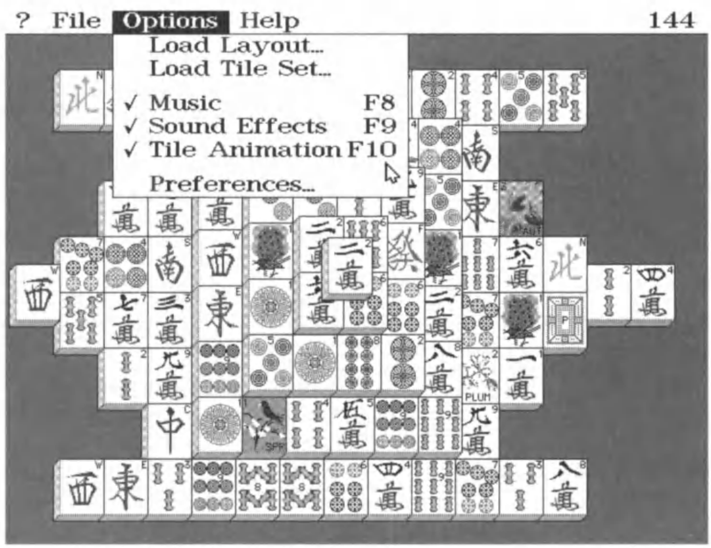
Although Shanghai is a fascinating game that can keep you captivated for hours, the first PC version was a disappointment because of its poor quality CGA graphics.

Until a program called Mahjongg appeared on the shareware market, the PC didn't have a comparable game that was worthy of its abilities.

Excellent version of Mahjongg

However, soon Activision made up for Shanghai's shortcomings with Shanghai II. The result is an excellent version of Mahjongg.

In Shanghai II you can select different stones and even change their layout on the board. An interesting aspect of Shanghai II is its mastery of digital sound, which can even be played over the PC's internal speaker. With an AdLib or Sound Blaster card, you can also play music in the background.



Classic Shanghai

The computer responds to your moves with short animation sequences and appropriate digital sound. If you prefer, you can switch off these options.

Shanghai II is an absorbing thought game that should keep you entertained for hours.

2.2.2 Action and adventure games

Usually it's difficult to distinguish between action and adventure games because these games usually contain elements found in both types of games (i.e, action and adventure).

The following games are a good representation of what's available for both action and adventure games.

Prince of Persia

Action and adventure

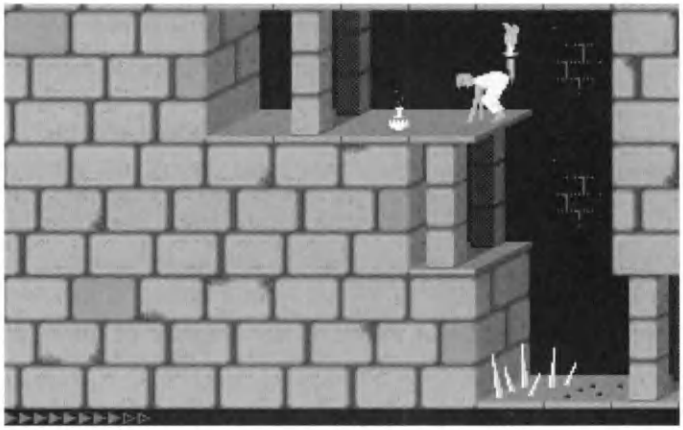
Prince of Persia is an example of an action game that also offers some adventure.

Imagine yourself as a young prince under the reign of a sultan in the distant Orient. With the sultan away, the Grand Vizier intends to seize power. To achieve his aim he must either marry the Princess or, if she refuses, kill her.

To prevent you from stopping him, the Grand Vizier throws you in the dungeon. From this moment on, you have 60 minutes to save the Princess. You race and struggle through 12 levels from the dungeon to the castle above.

The EGA and VGA graphics of this program are good. While the scenario of the game may seem unimaginative, the animation of its characters is excellent. You'll marvel as the Prince makes a mighty leap, clings to the very edge of the abyss, and slowly pulls himself up.

If you've played the game KARATEKA, such animation sequences may look familiar. The cast of characters for both games was created by the same programmer.



A typical screen from Prince of Persia

Prince of Persia was one of the first programs to deliver sound effects over the digital channel in addition to playing excellent AdLib music. These sound effects are some the best that have ever been produced.

TIP

If you start this program with the "PRINCE MEGAHIT" command, certain key combinations provide special help for your difficult climb.

Gods

When were gods not really gods? In antiquity, of course, when people could still interact with the gods. The game Gods, from Bitmap Brothers, lets you return to that time.

In this game, you'll become an ancient hero, who is undefeated in battle, and therefore bored and looking for challenge. Since you can't find an equal rival among men, you accept the challenge of the gods to liberate a captive city from four fierce sentinels. If you succeed, your reward is immortality.

You move over platforms and ladders, jumping, scrambling, and fighting. The VGA, EGA, or Tandy graphics scroll by in all four directions. The large sprites that move across the screen are worth seeing, although in VGA mode only 16 colors are used. All this makes Gods on of the few a jump-and-run games available for the PC.

But there's more to this game than tearing around mindlessly and shooting in every direction. To master every level successfully, you

must also solve a few puzzles. Which switches on the wall do you have to activate, and in what order? Which weapons should you purchase from the dealer you happen to meet?

So Gods offers a combination of pure action and strategy. Your goal is to vanquish the four sentinels. For each victory, you must traverse three levels, until you finally meet your ultimate enemy.

Your saga is accompanied by music played over the FM voices of the sound card. Don't miss the interesting tidbit the programmers have slipped into the title screen.

Compared to the AdLib version, the Sound Blaster version contains a pleasant, but subtle, difference in the title music. Some of the instrumentation has been digitized and is played over the digital channel. With AdLib, only the FM chips are used. Listening to them both provides an interesting comparison.

Gods is one of the first programs to synchronize the use of FM voices, digital audio channel, and graphics effectively.

This game is perfect for game fans who feel they've been missing real action on the PC. A 12 MHz AT is recommended to handle the high-speed graphics. Also, to get the feel of the action, hook up a digital joystick. Some players have scored over 2.5 million points this way.

Stellar 7

Solid-filled vector graphics

Stellar 7 is a game that first appeared on the C64. The new version has been improved. On the C64, this game provided only wire mesh models for game objects. However, the PC version of Stellar 7 offers solid-filled vector graphics in 256 colors, which run fast enough even on a normal 12 MHz AT.

We don't recommend using EGA and CGA graphics for this program because too much of the atmosphere is lost.

The concept of the game hasn't changed at all. You've been chosen to save the world from the evil Draxon, who wants to rule the universe. You have a war machine loaded with every kind of weapon.

The fighting takes place on the surfaces of planets you're trying to save from the enemy war machine. Before your mission, you can study the characteristics of each type of opponent you'll face.

A sound card is an excellent enhancement for Stellar 7, and with a Sound Blaster you can even enjoy digitized speech. This makes Lord Draxon even more frightening.

You can continue your adventures in a sequel to Stellar 7, called Nova 9. This program is another release from Dynamix.

Wing Commander

Most computer game fans are familiar with Origin's Wing Commander II and its popular predecessor, Wing Commander I. Owners of other computers have admired this game and have waited for it to be available on their systems. This interest from competitors is very unusual for a PC action game.

Instead of solid-filled vector graphics, Wing Commander's technology is based on bit-mapped graphics that are enlarged or reduced as the distance from an object changes. In other words, an object isn't composed by combining planes (surfaces). Instead, it's drawn from a single graphic in sizes that change constantly because of the calculations of a complex algorithm.

This process obviously requires a lot of computing time, so Wing Commander cannot be fully operational with less than 2 or 4 Meg of extended memory on a 16 MHz 386. It also occupies a lot of hard disk space. You should have 16 Meg available for the complete installation of Wing Commander II. Once this is combined with your Sound Blaster card, you're ready to enter the cosmos.

In the first part, you must accomplish several missions in your spaceship to preserve mankind from the evil Kilrathis. Starting out with a small machine, you can quickly work your way up to glory, fame, and a faster spaceship.

Your missions begin on board the Tiger's Claw, a huge spaceship, where you can also chat with your co-pilot or the friendly bartender.

The element of humor distinguishes Wing Commander from a simple space-shooters game. Also, the outstanding animation sequences seem more like an interactive video than an action game.

At the end of Wing Commander I, the evil Kilrathi swears revenge

Wing Commander II also uses another element to give the game a whole new dimension. This is the digital speech output, which you can enjoy only if you have a Sound Blaster card.

There is a terrific intro, in which the Emperor of the Kilrathis is speaking with the commander of his fleet. In clear dialog, you hear that the Tiger's Claw has been annihilated and that you're being blamed, because the human fleet command is unaware of the Kilrathis' ability to make their ships invisible.

A sequence from the WC II intro

Then the plot of Wing Commander II starts to unfold. You've been demoted for the loss of the Tiger's Claw and sent to a remote outpost. For a long time you've barely seen a patrolling spaceship or been aboard one.

However, things soon change. Determined to prove your innocence, you set off on your own against the Kilrathis. As in the first part, your skill and prowess are challenged along the way. Chance

plays a part, but ultimately the success or failure of your mission is up to you.

If you have the necessary hardware, Wing Commander is a fantastic game, and Wing Commander II, with the addition of a speech diskette that can be obtained separately, is enriched even further by dialogs during the battle missions and interludes.

Boarding your spaceship

Both parts of Wing Commander are impressive, but only Wing Commander II, with its use of digital speech, uses the unique capabilities of the Sound Blaster card. Even the AdLib music in this game is among the best game music around.

It blends as smoothly with the action as background music in a film. At times it dominates, sweeping you into its gripping tempo as the excitement of the story escalates.

At times it's so hushed and subtle you barely notice it, although you'd definitely miss it if it wasn't there. Wing Commander provides a masterful example of sound in a computer game.

Space Quest saga

For science fiction fans

If you're a science fiction buff, you'll enjoy the Space Quest adventures from Sierra. Although the story is suspenseful, it's also full of humor and allusions to science fiction.

SQ I

In Space Quest 1, you're a deck swabber on board the Arcade. The ship is transporting the star-generator, which is a machine that can transform planets into suns. Your people want to make a new sun for a dying solar system.

However, they haven't reckoned with the terrible Vohaul. His henchmen overtake the Arcada, kill every crew member they can

find, and steal the star-generator. By accident you survive because you happened to be in the storage closet.

Much to your distress, the Arcada's self-destruct mechanism has been activated. So there's only one thing you can do: Bail out in an emergency capsule before the ship explodes.

Crash-landing on an unknown planet, you now must endure treachery and peril until you reach the town of Ulance Flats. There you learn the whereabouts of the star-generator. You obtain a new spacecraft and immediately fly off in pursuit of Vohaul's ship.

Trying your luck in the bar at Ulance Flats

Now you must get to the star-generator undetected and destroy it.

When you succeed, you return to earth a hero and get a new job: swabbing the decks on a new spaceship.

SQ II

In Space Quest II, you've started your new job. Suddenly you're knocked unconscious and abducted by strange men.

As you regain consciousness, you figure out that Vohaul was behind your kidnapping. He hasn't forgiven you for the loss of the star-generator and intends to have you killed.

You've just settled in at your new job...

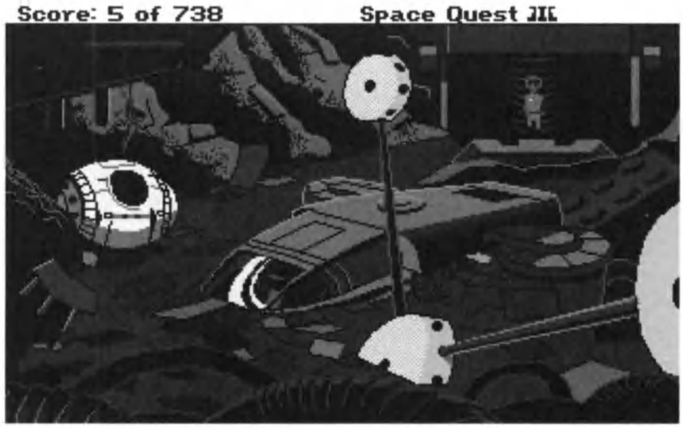
Fortunately, his henchmen aren't too intelligent. So on the way to your execution you manage to escape into the jungle. Unfortunately, you'll encounter innumerable dangers there.

You emerge in death-defying triumph over all these dangers and manage to find another ship, in which you sail back to Vohaul's asteroid and demolish it. Finally, you escape in a capsule and fall into a deep sleep.

SQ III

Space Quest III begins as you're drifting aimlessly through the cosmos and are picked up by a scrap freighter. Shaken by the commotion, you awaken from your sleep.

Ingeniously, you build a functioning spacecraft from the scrap inside the freighter and blast off to safety.



You've found a ship you can repair

Later, at an intergalactic fly-in, you discover a hidden call for help from the "Two Guys from Andromeda" in a video arcade. They have been abducted by a game developer and are being forced to program senseless arcade games.

Resolving to rescue the unlucky fellows, you set off on a journey that takes you to several planets. You must gather paraphernalia and information along your way. However, you must be careful because not everyone you encounter has good intentions.

In the end, your mission is successful and the two guys are saved. You take them to a famous software firm where they can program happily ever after.

But, unfortunately, you're out of work again. What are you to do, an unemployed swabbie and hero?

SQ IV

Space Quest IV begins with you hanging out in an intergalactic pub, telling tales of Vohaul and other formidable adversaries. Suddenly the police arrive and escort you from the premises.

Soon you learn why this happened. Vohaul is still alive, and this time he really wants you dead. He's fed up with all these sequels and the whole Space Quest saga. That's why he's sent the Sequel Police after you. Only a space-time warp from a Time-ripper manages to save you from a horrid and gruesome fate.

Now a series of really strange adventures begins. You meet people from your future who aren't happy about the things you'll do to them. This takes you to Space Quest XII.

Vohaul is still following you in Space Quest XII

When you return to Ulance Flats, things are different than you remember from Space Quest I.

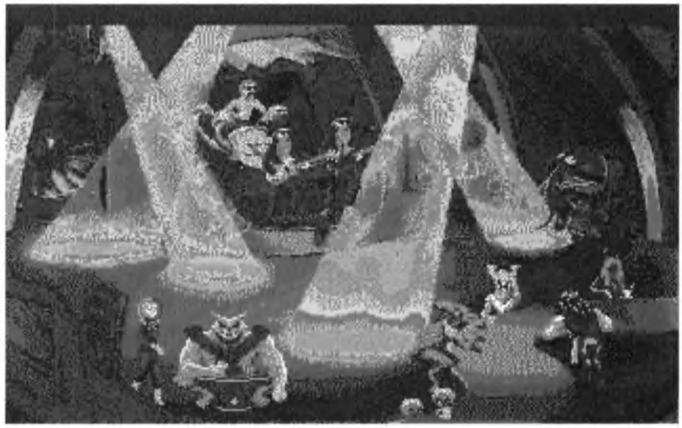
Ultimately, however, by successfully negotiating all the perils and puzzles of this adventure, you can arrive at a happy ending. We'll let this part of the story be a surprise.

The original releases of Space Quest I and II use EGA graphics and lack sound card support. Starting with Space Quest III, you can finally hear the Space Quest theme played with AdLib quality.

Part IV offers additional improvements. Besides the EGA version, a separate VGA version can be purchased. Both releases use the Sound Blaster card's digital channel.

Although Part IV offers excellent graphics and sound effects, the plot has suffered slightly.

If you're new to the Space Quest saga, you should purchase the updated release of Space Quest I. This is a VGA version that uses the Sound Blaster card just as Part IV.



A scene from Ulance Flats in VGA

So if you like your adventure games to have some humor, check out the Space Quest saga. Its creators poke fun at both science fiction and themselves.

Star Trek 25th Anniversary

Action and adventure

In this game, you're transported to the year 2200. The game involves the adventures of the Starship Enterprise, with its 400 member crew. For five years the Enterprise has been discovering new worlds, new life, and new civilizations. Lightyears from earth, the Enterprise presses onward into galaxies no man has seen before.

Star Trek 25th Anniversary didn't arrive on the market until the 26th anniversary of Star Trek. The game combines both action and adventure.

Your role is of James T. Kirk, captain of Starship Enterprise.

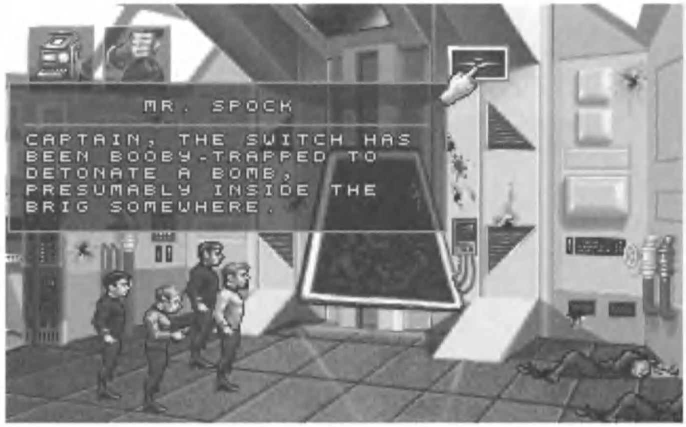
The bridge of the Enterprise

The sound effects don't use the Sound Blaster's digital channel, although the FM voices can create excellent effects. The familiar Star Trek theme is played before each mission. A digital version of the original music would be even better. With a Sound Blaster Pro you can even hear the music in stereo while playing the game.

The game itself takes place in two areas. From the bridge of the Enterprise, you conduct your conquest of space, while adventure awaits you on alien ships and planets.

Aboard the Enterprise, you have the support of Uhura, Scotty, Chekov, Sulu, and Mr. Spock. You order Scotty to take charge of other areas of the Enterprise, or consult the ship's computer for crucial data. You even obtain information about members of the Enterprise crew.

Dr. McCoy, Spock, and another officer of the Enterprise accompany you to planets or other spacecraft. Just like in the TV series, this officer is usually destined for a hero's death.



Liberating the prisoners aboard the USS Masada

Seven missions allow you to prove your prowess as a true "Trekker."

Similar to Wing Commander, this game could easily be augmented by supplemental diskettes containing new missions. However, no such plans have been announced. If you like space games and enjoy the television series, you should enjoy Star Trek 25th Anniversary.

Ultima Underworld - The Stygian Abyss

Role-playing game

You may be familiar with this series of role-playing games that are based on the land of Britannia. In this seven part series, players prove their bravery at the side of Lord British, as he defends Britannia from danger on every front.

The graphics in this series have kept pace with recent technological developments. So, the animation in Ultima VI and VII is among the best in the field of PC games.

In other role-playing games, you click arrows to guide your game figure through passages and tunnels. But in Ultima Underworld *you* move through the passages and tunnels. As floors, walls, and ceilings roll by, you actually experience the sensation of movement. You can look up and down and turn to any angle. The program adjusts your view of the surroundings accordingly.

With practice, you'll get used to running around corners and jumping over crevices. An auto-mapping function provides an overview of the action.

The type of control you have in this program and the speed with which the "world" reacts to it is similar to "cyberspace." You venture into the fabulous Underworld of Britannia and travel throughout it at will, talking with friends, finding needed objects, and clashing with fierce monsters.

Objects in the Underworld, as in Wing Commander, are created with bit-map graphics, which are enlarged or reduced as the objects "move" with respect to the observer.

Ultima Underworld can serve as a model for other role-playing games. With it, Origin has again set a new standard for PC software.

Another similarity to Wing Commander is the use of Sound Blaster's digital sound capabilities. The preview animation introduces the player to the object of the game in an entertaining way. However, later the player experiences the underworld of Ultima, which isn't so entertaining.

The FM-voice music plays in the background; this music is played in stereo on Sound Blaster Pro.

Ultima Underworld is a whole new sensation in role-playing. However, its graphics and sound are very demanding on your hardware. You should have at least a 386 computer to play it at high detail. Reducing the detail of the Labyrinth World makes the program run a little faster, but you sacrifice some of the impact.

Also, you need 13 Meg of available hard disk space to install the entire program. If you're willing to forfeit some of the intro animation and the speech audio, you can decrease this amount by almost 4 Meg. This might be a good idea once you've seen the introduction a few times. However, you should install the complete package for at least one viewing.

Galactix

Space combat game

If you like space combat games, you should enjoy Galactix.

This game presents the following scenario: From an ecological standpoint, the earth is breathing its last breath. You have just seen a newscast reporting that the last tree in the rain forest has fallen.

The rain forest is lost forever

Suddenly the TV broadcast is interrupted with a picture of an extraterrestrial being who informs you that you have been conquered by "Xidus" and are about to be enslaved.

The aliens are at your door

You have only one chance to save the world. A specially-outfitted space fighter awaits your command against the aggressor Xidus. Now the game begins.

This is the best version of Space Invaders that's available on the PC, and technically it's very impressive.

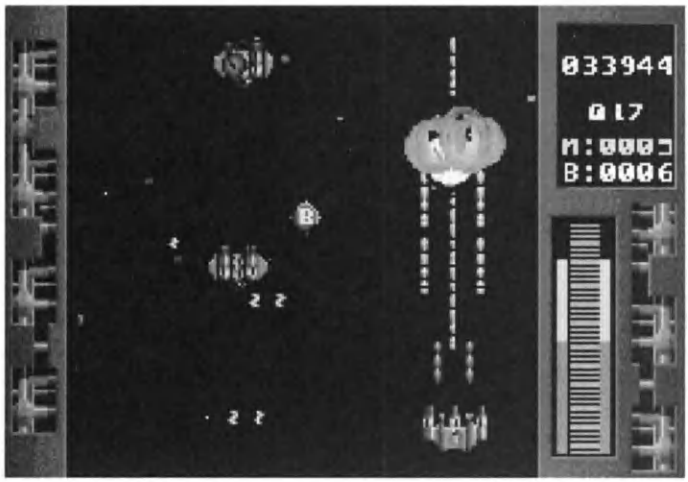
The game offers pure VGA graphics in 256 colors. The full impact of the sound effects is realized only with a Sound Blaster or compatible sound card. All the text in the intro is played as

digital speech and, during play, sound effects come over the digital channel, while excellent FM music continues to play.

Very few programs offer such a rich blend of audio enhancements. The resulting "carpet of sound" adds a new and thrilling dimension to the game.

The game itself is also well-programmed. However, the possible variations are somewhat limited, as with most combat games, but all the elements of Galactix fit together well.

As the battles rage, you're able to accumulate accessory weapons, rockets, and bombs to fortify your craft.



A typical Galactix scene

You fight your way from planet to planet, moving deeper into the solar system. On a 386 machine, Galactix provides over 100 playing levels.

Perhaps the best feature of Galactix is that's it's a shareware product. Registration with the software firm Cygnus costs only $15. According to the registration terms, part of this money goes for conservation of the rain forest.

2.3 Shareware and Public Domain Software

Shareware and public domain

When the Sound Blaster card was first introduced, most shareware and public domain software didn't support the Sound Blaster card. However, as the card was improved and, simultaneously, the price

decreased, more users became interested in the Sound Blaster card. At this time, many users discovered that the programs they needed for specific purposes weren't available. So some of theses users wrote their own programs.

As a result, a few programmers, such as Mark Cox and Gary Maddox, are well-known in the field of Sound Blaster programming.

In this section we'll discuss some of the best shareware and public domain programs that have been developed and where you can locate these products.

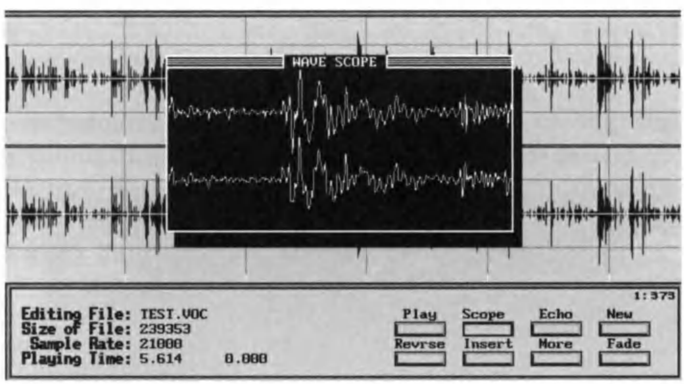
VOC tools

This category includes programs that work with various types of sound samples.

Blaster Master

Supports stereo capabilities

The Blaster Master program by Gary Maddox was one of the first to enable editing of VOC files on the screen. Version 5.0 is the most recent version, which provides all the functions you need when working with sample files. Even the special features of Sound Blaster Pro are supported.



Version 5.0 supports SB Pro's stereo capabilities

You can read, play, and record files. On-screen functions let you add or remove sections, alter the speed or volume, add echo, fade files in or out, and even mix two files together.

Blaster Master does all this, not just with VOC files, but also with WAV (i.e., Microsoft Windows format), SND, and NTI files. The

SND format is usually a pure data format without headers. NTI (Noise Tracker Instruments) files are a special Amiga format used with Amiga composer software. These formats can also be interconverted.

Unpacking packed files

A special feature of Blaster Master is its ability to unpack packed sample files. This allows you to then work with the files normally. Both Blaster Master and Voice Editor from Creative Labs cannot perform most editing functions on packed files.

However, the quality lost in packing cannot be reclaimed. Blaster Master simply "stretches" the packed bytes back to their original 8-bit length. The information deleted by the packing process isn't retrieved.

With the shareware version of Blaster Master you're limited to editing files of up to 25 seconds of sample data. This limitation doesn't exist in the registered version.

Blaster Master is definitely competition for Voice Editor. In fact, at one time you could even get a registration allowance by sending Gary Maddox your Voice Editor in payment.

If you don't own Voice Editor from Creative Labs, Blaster Master will give you a comparable, in some ways even superior, tool for editing VOC files.

Sputter Sound System

Interface allowing you to play different sound formats

The Sputter Sound System is an interface from which you can play files, with completely different sound formats, over the Sound Blaster card. It can handle the various formats of the Covox Speech Thing, Amiga IFF, Amiga Sonix, Macintosh Sound files, and, of course, the VOC format. Even a few other formats are supported; some of these are very rare.

AdLib ROL files can also be played from the Sputter interface. Sputter finds the AdLib driver SOUND.COM, starts it, and then starts the song.

Sput Text-to-Speech

Since Sputter Sound System also has a speech synthesizer that can translate text into speech, you don't have to own SBTalker in order to use this program. Although the results aren't comparable in quality to SBTalker's, they are understandable.

You can even tell Sputter how to pronounce words by using a lexicon. Text can also be output phonemically in writing to verify

pronunciation. Spoken text can even be output to a file instead of to the speaker.

Sput Event Monitor

One function of the Sputter Sound System that you don't use from the interface is called the Sputter Event Monitor. This is a small memory-resident program that constantly monitors your computer for a certain triggering event, which causes it to play a sound.

A certain key could also trigger a sound, for example a confused call for help, when you press F1. System interrupts can be interrogated just as easily. Your imagination is the only limit.

The registration for Sputter Sound System costs $25. With it you receive instructions on how to make the shareware version into a full version. The full version then skips the shareware notice at the beginning of the program and after playing sound files.

CD-Box

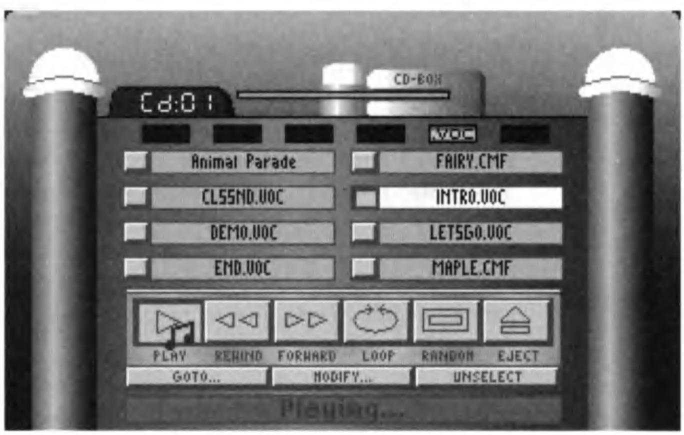
Similar to the Sputter Sound System

CD-Box is program that's similar to the Sputter Sound System.

Suppose that you have files of different formats to be played. For each one, you must call the proper play program (VPLAY.EXE for a VOC file, MP.COM for a MOD file, etc.).

The CD-Box makes it a lot easier. From this program you can play any VOC, MOD, CMF, ROL, or MUS file, if you have the appropriate play programs.

So, the CD-Box acts as an interface for calling the various utilities. It's implemented in VGA graphics with 256 colors.



The CD-Box menu

The author, Jeffrey Belt, has also inserted a few fun tricks, such as a spider occasionally dropping down from the top of the screen. An excellent graphic animation sequence puts on a record when CD-Box plays a recording.

However, one disadvantage of this program is that all the music files must be in the same directory as CD-Box. To keep your files organized, you can create a ZIP archive for each type of music, which CD-Box then scans automatically.

If you listen to a musical piece only occasionally, you probably won't need CD-Box. But this program is perfect for playing different music formats under a single interface.

Remember that CD-Box doesn't work flawlessly with Version 2.19b of Modplay yet. But even if you only use it to play your numerous VOC and CMF file archives, the program is worth having.

CD-Box is public domain, so there is no registration fee. A future version has been announced and possibly will no longer require external play programs. This would make CD-Box the most universal play program for the Sound Blaster card.

WavePool

Creates VOC files by programmed calculation

A completely different program for VOC files is WavePool. The name is an abbreviation for "WAVE Programmable Output Oriented Language".

In this program, VOC files aren't created in the usual manner (i.e., digitally). Instead, they are created by programmed calculation.

WavePool originated with the idea of creating sound effects for Windows. The result is a programming language that closely resembles C in its structure. Most of the same mathematical functions and commands, such as "printf" or "do/while", are available. You can even define your own functions.

Simply write an appropriate program with a text editor and let WavePool translate it into a VOC file. The standard sampling rate is 11,000 Hertz. An example of a WavePool program is shown in the following short listing:

```
#
#  Demonstrations-Sound
#      for Wavepool
```

```
#  (C) 1992 Data Becker GmbH
#  Author : Axel Stolz
#

BEGIN {
   screate("LASER.SOU")               # Create sound file

   for (x=700; x>=100; --x)           # Loop from 700 to 100
{
      Hertz = exp(x/120);             # Assign EXP function
to variable
      print "Frequenz:", Hertz;       # Output variable
      sinwave(Hertz, x/5);            # Generate sine wave
      quiet(5);                       # Add 5 Bytes quiet
}
   sclose(0);                         # Close sound file
   sndvoc("LASER.SOU", "LASER.VOC"); # Convert sound file
}
```

For copyright reasons, the sound that this program generates isn't included on the companion diskette. Registration is first required. The source code file, however, is on the diskette under the WAVEPOOL directory.

Although WavePool was first intended for Windows, files cannot yet be saved in WAV format. For this you still need the Creative Labs program VOC2WAV. But this will probably change.

The registration for WavePool costs $35. This gives you the current program version and authorization to transfer sound created with WavePool. The copyright is held by Data Assist Inc. of Columbus, Ohio.

Converting samples

Because there are so many different sound programs, there are also various file formats for samples. To be able to use your favorite sound in your favorite program, you may need a utility that can convert the sample data to a different format. Several of these utilities are available.

IFF2VOC, SUN2VOC, and VOC2SND

IFF conversion

The first two programs, from Kevin Bachus, convert an IFF sample file and a SUN sample file, respectively, into CT format.

IFF samples are found mostly on the Commodore Amiga. The header contains the sampling rate at which the sound was

recorded. Therefore, you don't have to worry about anything when you convert a file. IFF2VOC handles everything automatically.

SUN2VOC is very similar to IFF2VOC. This program converts Sun-Workstation files into VOC format. Since SUN samples don't always have a header, the standard sample rate is 8000 Hz.

SND conversion

The program VOC2SND, by Kevin Withnall, works in the opposite direction. With this program, a VOC sample is converted to a SND file, which is the usual Macintosh format. However, currently it only recognizes data blocks. So it ignores the extra refinements that the CT format offers.

Sound Exchange - Shareware:SOX

Converts in either direction

SOX can convert files of several formats (IFF, SUN-Audio, Macintosh HCOM, RAW, IRCAM, and VOC) in either direction. Effects, such as echo and vibrato, can also be added.

SOX originated on Unix computers, but because of portable C source codes there are also versions for MS-DOS. So, this program can be obtained only via Internet.

Soundtracker programs

Impressive musical segments

One music program written for the Commodore Amiga has virtually created a standard of its own. This is the Soundtracker program, which uses digitized instruments to create very impressive musical segments. Soundtracker music sounds much more realistic than any created through sound synthesis.

The Amiga has four audio channels, two of which are used for the left stereo channel and two for the right. On a Sound Blaster Version 1.0, 1.5, or 2.0, all four must be output to a single channel, resulting in loss of the stereo effect.

Some MOD players, as the play programs for Soundtracker modules are called, can separate the output channels on Sound Blaster Pro to use its stereo capability. As a further advantage, they have adopted the Amiga's data format completely, so you can use all the Amiga modules without having to convert them back and forth.

In this section we'll presents a few exceptional MOD programs.

Modplay Pro

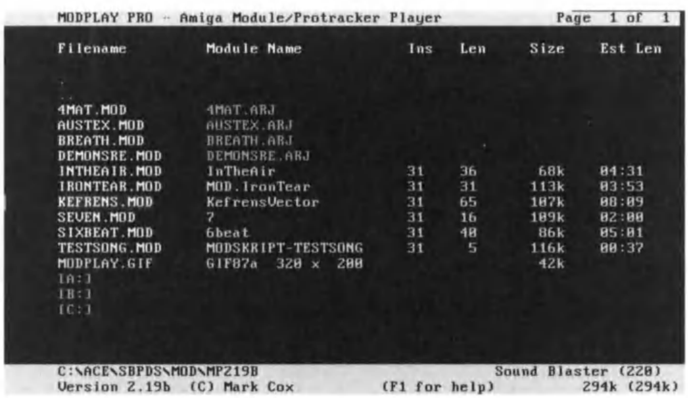
The Modplay program can play Soundtracker and Protracker modules. Protracker is one of the latest Soundtracker emulators on the Amiga, with a few extras of its own.

Flexible module output

The current version 2.19b of Modplay provides extremely flexible module output. Besides the Sound Blaster card, you can choose between various D/A converters and even the PC speaker. A DOC file included with Modplay even contains the assembly instructions for an inexpensive D/A converter on the parallel port.

Another positive feature of this program is that MOD files, which can occupy huge amounts of storage space on your hard disk, don't necessarily have to be in unpacked form. Modplay will scan LHArc, PKZip, Zoo, and Arj archives for MOD files. So you can put all your modules in one archive, and Modplay will unpack the desired file as needed.

You can control Modplay by using command line parameters or by selecting the interactive mode. In this mode, you can use a directory list to scan your hard disk for MOD files. A simple keystroke starts the corresponding file.



File selection in ModPlay

While a song is playing, certain information can be displayed. First you have the option of displaying a GIF image in the background, while an oscilloscope is displayed in the foreground.

Then, you can have individual patterns (i.e., notes and effects within songs), displayed as text, or simply display the name of the instrument that's currently playing.

The realtime spectrum-analyzer is especially useful. The program's author, Mark Cox, programmed an actual spectrum-analyzer here. However, since it occupies a lot of computer time, it's actually "realtime" only on faster machines (386 or higher).

During play you can adjust the song's volume, skip over patterns, and speed up or slow down the playing speed. The four channels can be switched on and off individually. A useful feature lets you switch to DOS, during play, to look at a directory, for example, while Modplay continues in the background.

Another interesting option allows you to write a list of all your modules to a file. This can be helpful for locating duplicates.

In interactive mode you can also save individual instruments from a module to separate files for use in other programs.

Modplay is public domain with no registration fee. For 5 British pounds you can get the latest version directly from the author.

ModEdit

Create your own MOD files

ModEdit is a program you can use to create MOD files yourself. ModEdit's author uses a memory-resident version of the program Modplay, by Mark Cox, to play back the music created from within the editor environment. So you can access the same output devices as under Modplay.

A Soundtracker module consists of up to 128 different patterns that are combined to make a song. Each pattern contains 64 notes for each of the four channels.

In Chapter 5 you'll learn more about the Soundtracker format. There is also an example program to write these patterns from a Soundtracker module to a text file.

ModEdit can also load any module and show its patterns.

However, the main purpose of this program is to create patterns to compose a song. You enter the notes one by one and then manually transfer them from a notebook, for example.

It's also easy to compose with ModEdit. To hear how the notes you've just entered sound together, you can play the piece back at any time. ModEdit recognizes the usual Soundtracker effects commands, such as Volume Slide and Pattern Break.

If you want to be creative with Soundtracker modules, ModEdit is a useful tool. The finished product can be played on any MOD player.

Author Norman Lin has given ModEdit to the public domain, so there is no fee for registration. However, he does accepts donations of $5.

Scream Tracker

New MOD format

A new MOD format that's widely used on the PC is the Scream tracker format. Scream Tracker is a professional software product with a shareware version.

In the shareware version, you cannot play modules or read Amiga modules. The layout of the Scream tracker module differs in a few areas from the original Soundtracker format.

In Scream Tracker you can enter notes by playing them, as on a keyboard, or by manually inputting their values. There is a convenient clipboard where you can place part of a pattern, for example, in order to copy it to another place.

Your instruments can be on up to 99 different diskettes, which Scream Tracker manages. This is very similar to the Original Amiga Soundtracker concept.

During playback, you can track individual patterns to weed out wrong notes. An oscilloscope is also provided, as, for example, in Modplay.

Unlike ModEdit, Scream Tracker is not fully functional in the shareware version. Upon registration you receive a version in your own name that allows you to save your modules. The fee is $48. In addition to the latest version, you receive a Quick Reference Card and a selection of Scream Tracker songs.

CMF programs

Few programs available

The shareware and PD markets offer only a few programs in the CMF area. The ones that are available are mostly player routines rather than composition programs. One exception that allows you to express your creativity is a program called Compoz.

Compoz

Although Compoz is mainly a composition program for files in Adlib ROL format, it finally provides a way (more or less) to generate CMF files directly.

Although internally the program works in its own CPZ format, you can directly read and write CMF files. The SBI format is similarly supported. We'll discuss these formats in more detail later in the book.

Notes are designated in the standard musical notation, which is commonly used by composition programs. Since Compoz works in text rather than graphic mode, the notes look a little unusual, but you soon get used to it.

Compoz supports the 9-voice mode of the Sound Blaster card. These voices can be selected from 16 available instrument sounds.

Although notes can be entered with the mouse or the keyboard, the mouse is the better alternative.

With a little practice and some familiarity with the notation system, you can start computerizing your own musical creations.

The shareware version of Compoz is fully functional, so it costs you nothing to see how it works. Registration is $20.

Perhaps other programmers will follow Compoz author John M. Coon's lead and develop more programs for creating CMF music.

2.4 Finding Public Domain and Shareware

One way to obtain Sound Blaster software is through shareware and public domain vendors. These vendors usually provide catalogs for free or for a nominal fee which list their selections. You can then buy diskettes containing the programs from them, but you may still pay a registration fee to the programs' authors (for more information about shareware, refer to Abacus' *Finding (Almost) Free Software*).

Bulletin Board Systems (BBSes)

If you have a modem and terminal software, public domain and shareware software is a phone call away. A Bulletin Board System (BBS) is a computer running BBS software and connected to a modem. This BBS (or host) computer can then be used by other computer/modem users for exchanging messages, ideas, programs, and even electronic mail.

Many BBSes feature Sound Blaster support. Creative Labs has their own BBS, which, at this writing, can be reached at (408) 426-6660 (call Creative Labs' voice phone number FIRST, to check on the validity of this phone number before dialing the above BBS number with your modem).

As with any BBS, you pay for the telephone call, and some BBSes may charge membership fees–understandable when you consider how much time and effort a system operator (or SYSOP) will devote to maintaining such a communication link. Internet also provides access to Sound Blaster software, but most Internet users are part of a corporation, university or other large organization (more on this later).

Online services

You can consult paid online services such as Prodigy and Compuserve Information Service. For example, Compuserve has such Sound Blaster software as sound samples and utilities in its MIDIFORUM area.

Internet

International computer association

You may have access to the international computer association Internet. Many large universities and mainframe installations provide this service.

Internet's software selection is huge. Besides mainframe software, you'll find support for the PC, Amiga, Atari ST, Apple Macintosh, Apple II, NeXT, and Unix computers. In an Internet node that offers only Amiga software, for example, you might find the latest Soundtracker modules to download for playing to your Sound Blaster card.

These Internet nodes already have several categories with many listings for Sound Blaster and AdLib software.

If an Internet address is changed, this usually isn't a problem because almost all Internet nodes contain lists of addresses for particular subject areas.

The following list contains Internet addresses that relate to the general subject of sound. This should help you get started in your search for suitable software. Individual directories of the Internet nodes sometimes contain index files showing the subdirectories where programs are located.

Selected Internet addresses:

ccb.uscf.edu

Sound-Player, SND files

cobalt.cco.caltec.edu

Sound Blaster Programming

ftp.ee.lbl.gov

SUN-Audio files

garbo.uwasa.fi

Sound-Converter, Scream-Tracker files, Sound-Player

nic.funet.fi

Sounds, tools, information

saffron.inset.com

Comprehensive collection on Sound Blaster and Adlib, but often very slow access

snake.mcs.kent.edu

Sound Blaster and Adlib programs

ucsd.edu

MIDI information

vaxb.acs.unt.edu

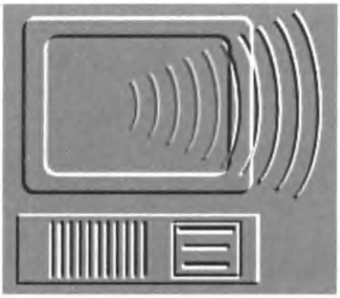
MIDI information and Internet library listings

wsmr-simtel20.army.mil

Various Sound players

xanth.cs.odu.edu

Sounds, special for Star Trek fans



Chapter 3

Windows Software

Sound Blaster and Windows

When Microsoft Windows Version 3.0 was released, it included support for various graphics adapters. However, even this version didn't support any sound cards.

Although many graphics cards had entered the PC market, none of the available sound cards were widely used.

This situation began to change with the appearance of the AdLib sound card. As AdLib and its early rival Sound Blaster became more popular, enthusiastic programmers tried to incorporate their voices into the Windows environment. As we mentioned, the earliest utilities were written privately.

Microsoft responded to these events by developing its Windows 3.0 MultiMedia Extension. This package allowed the integration of various audio and video hardware extensions and included support for AdLib and Sound Blaster.

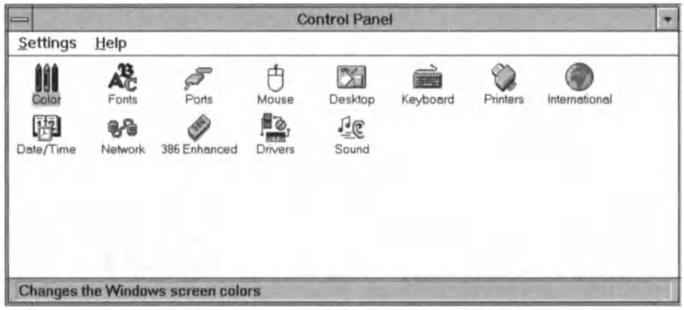
Later, with a few small exceptions, the functions of MultiMedia Extension were included in a new release of Windows Version 3.1.

Although the concept of multimedia encompasses more than just sound, we'll discuss only the audio aspect.

3.1 Installing the SB Sound Drivers

Installing the SB sound drivers

When you install Windows 3.1 on your hard disk, you won't notice any new audio features immediately. To bring your Windows sound system to life, start by opening the Main group and clicking the Control Panel icon.



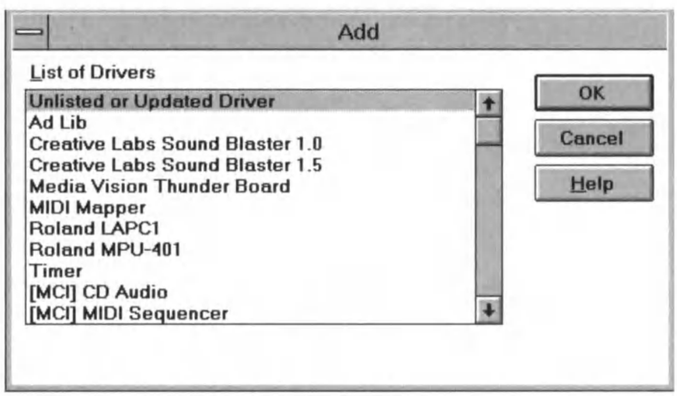
Control Panel

If you're used to working in Windows, you're probably familiar with the Control Panel. This is where you control the screen appearance and other peripheral interfaces of your Windows system.

Control Panel lets you choose colors, background patterns, and fonts. You can also select printer, keyboard, and mouse settings. A new icon, called Drivers, provides access to the world of Windows sound.

Adding drivers

When you double-click the Drivers icon, a list of drivers that have already been installed appears. To install a driver for your new Sound Blaster card, click the Add.. button. A list of available drivers appears similar to the following:



Driver selection menu

The available drivers include, besides AdLib and Roland, the Sound Blaster drivers for Versions 1.0, 1.5, 2.0 and Sound Blaster Pro.

Special SB Pro drivers

Select the necessary driver from the driver list. If what you need isn't listed, select "Unlisted or Updated Driver" from the driver list.

If you don't have the proper driver for your card, choose an earlier version from the ones available. Windows usually advises you to look for a newer version, but your installation can still continue.

Windows reminds you to get a new driver

HINT

Remember, an older driver cannot use the new capabilities of Sound Blaster Pro. For example, it won't recognize a different DMA channel number.

If you experiment with more than one driver, remember that each one you install occupies space on your hard disk.

After each driver installation, you're asked whether you want to restart Windows so your changes can be activated. If you must install additional drivers, it's faster to click on the [Don't Restart Now] button, install all the drivers you want installed, then click on the [Restart Now] button.

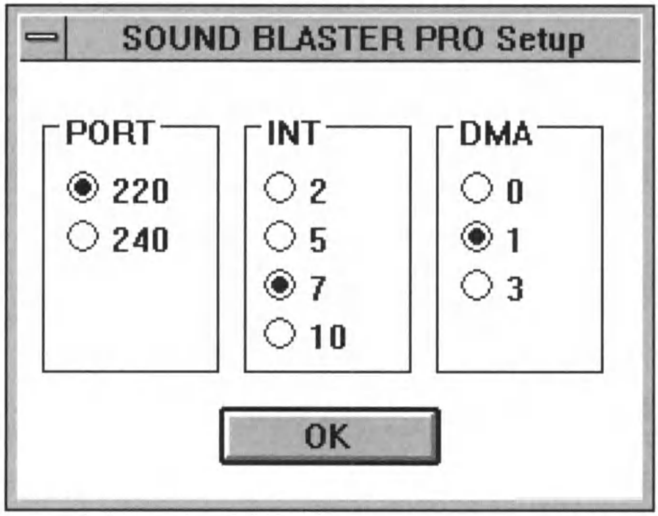
Using older drivers

The driver installation process lets you specify the parameters of your hardware device. Be especially careful when choosing settings for a driver that's not completely compatible with your card.

When installing the driver for Sound Blaster Version 1.0 or 1.5, you're asked for the port address and interrupt number. The selection list includes several port addresses. If your card is Version 2.0 or Sound Blaster Pro, only two of the port addresses (220 and 240) are valid. The interrupts also differ between cards. Refer to Chapter 1 for information on these parameters.

When installing a driver for Sound Blaster Pro, only the choices valid for that card are displayed.

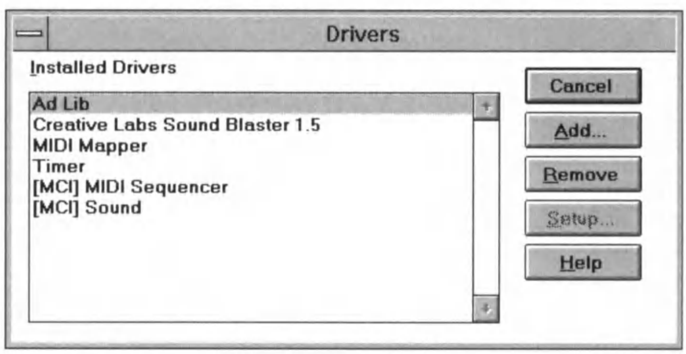
Sound Blaster Pro has three separate driver files that must be installed. Two ask for the port address only. The third also asks for the interrupt and DMA channel number.



The SB Pro driver considers the changed hardware

To play MIDI files under Windows, the MIDI Mapper must also be installed. We'll explain this in more detail later.

After you've installed some drivers, your screen will look similar to the following:

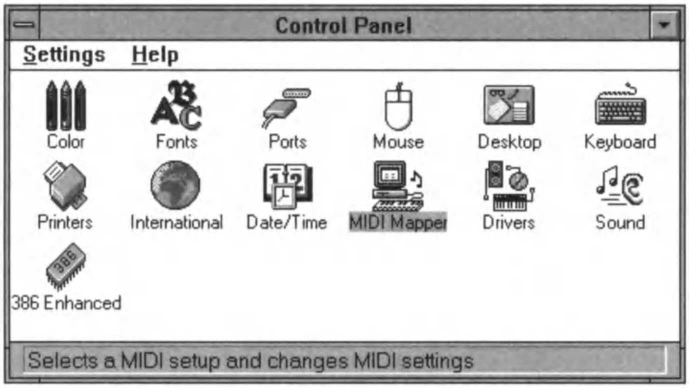


A series of installed drivers

Since it controls the FM voices the same way as Sound Blaster, the AdLib driver is added to the list automatically when you choose one of the Creative Labs drivers.

Now restart Windows to activate your new driver settings and return to the Control Panel group.

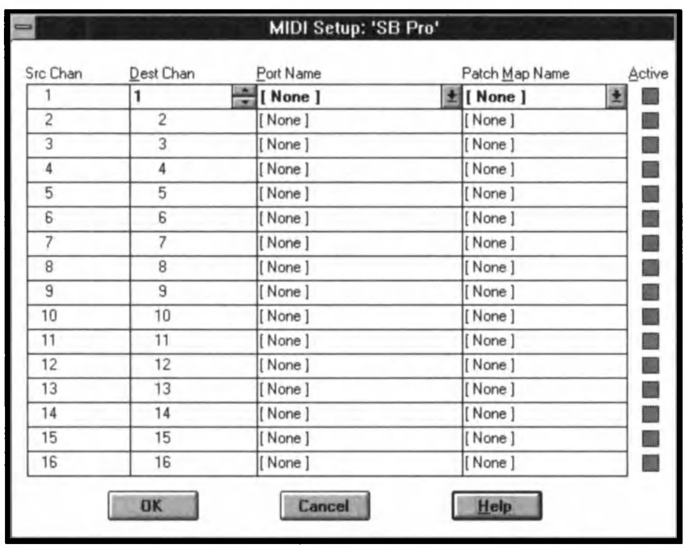
A new icon called "MIDI Mapper" appears:



The MIDI Mapper icon

Double-click the MIDI Mapper icon to open the MIDI Mapper window. If you don't have an AdLib card, you must create your first MIDI setup. First ensure that the Setups option button is selected.

Click [New...] and enter a name and description for your new setup (e.g., "SB Pro"). Click on the [OK] button.



Src Chan	Dest Chan	Port Name	Patch Map Name
1	1	[None]	[None]
2	2	[None]	[None]
3	3	[None]	[None]
4	4	[None]	[None]
5	5	[None]	[None]
6	6	[None]	[None]
7	7	[None]	[None]
8	8	[None]	[None]
9	9	[None]	[None]
10	10	[None]	[None]
11	11	[None]	[None]
12	12	[None]	[None]
13	13	[None]	[None]
14	14	[None]	[None]
15	15	[None]	[None]
16	16	[None]	[None]

You must create a suitable MIDI setup

You can now configure the Sound Blaster's MIDI capabilities. The sequence of source channels is preset from 1 to 16, but you can select the destination channels. Unless there is a good reason not to, you should also sequence these from 1 to 16.

Chapter 4 contains additional information on MIDI channels.

The Port Name entry specifies the output port to which the MIDI data should be sent. Available choices include the normal Sound Blaster card (Version 1.0 or 1.5), the AdLib card, Sound Blaster Pro in FM stereo mode, and the MIDI interface of Sound Blaster Pro.

If you don't have a MIDI synthesizer connected to the corresponding interface, select the appropriate Sound Blaster parameter for each channel.

When you're finished with the setup, click the [OK] button and click the [Yes] button to save the changes.

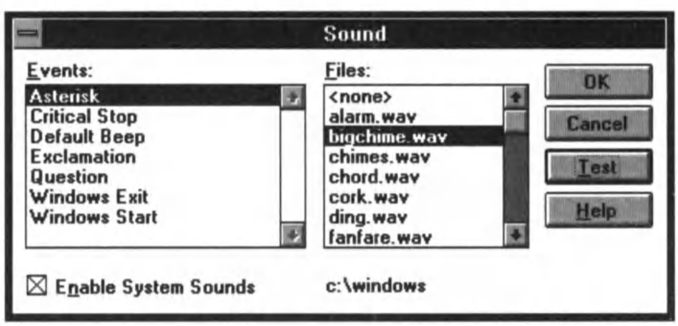
Most Sound Blaster Pro and Sound Blaster 2.0 packages include the necessary setups as files so you don't have to enter them yourself.

3.2 Sounds for Windows Events

Assigning sounds to Windows events

Once the appropriate drivers are successfully installed, you can start to make some noise. A quick way to get started is to assign sounds to certain events in Windows.

In Windows 3.0, the Sound icon was used only to enable or disable the warning beep played over the PC speaker. In Windows 3.1, it assigns different sounds to different events.



Assigning sounds to Windows events

It's still possible to switch the system sound on or off. But in addition to the familiar warning signal, a new repertoire of sounds is now available in the form of WAV format sound files.

These files can be linked to various events, such as Windows Start and Windows Exit, causing the sound to play whenever the event occurs.

If you're tired of the standard beep, try assigning DING.WAV from the Windows directory to the event called Default Beep.

Any WAV file can be assigned to any event. You can experiment to get the desired results. The previous illustration shows Asterisk linked to a file called BIGCHIME.WAV (a definite attention-getter).

The linking procedure is simple. Click on the desired event from the list on the left. Then click on the sound filename you want to assign to that event from the list on the right. To hear the sound at any time, click on the [Test] button.

Sounds in WIN.INI

If you look in the WIN.INI file after assigning sounds, you'll see that your selections have been saved as entries in the file's [sounds] section. You can do the same thing by editing WIN.INI directly.

These entries show the exact spelling of the Windows event names, which you can use in your own sound programs. We'll discuss this in more detail in Chapter 5.

3.3 Sound Recorder

Record your own sounds

If you want to add more sounds to your WAV file repertoire, simply record them yourself.

To enter your Windows recording studio, double-click the Sound Recorder icon in the Accessories group. Windows displays the following window:

Sound Recorder window

The commands on the menu bar allow you to open, play, and edit WAV sound files. The status of the current file (Stopped, Playing, or Recording) is displayed below the menu bar.

Under the status line is an area called the wave box. As a sound plays, the wave box displays the sound graphically like an oscilloscope. When a sound isn't being played, the wave box displays a flat line.

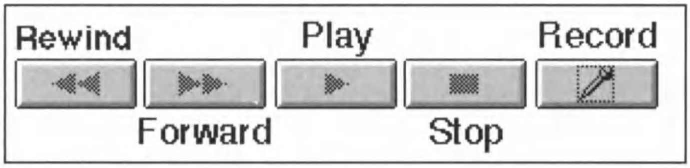
Your current position within the file is displayed to the left of the wave box, and the file's total length is displayed to the right.

You can easily see a sound's waveform

A scroll bar lets you move around within the WAV file.

Like a cassette recorder

The buttons at the bottom of the Sound Recorder look like the controls on a tape recorder or similar device. The first four (from left to right) represent the Rewind, Forward, Play, and Stop functions.



The last button represents a small microphone icon. This is the Record button. When you click it, Sound Recorder starts to record sound from the microphone input. Technically, this means that the program begins to read the data from Sound Blaster's analog-digital converter and write it to disk. Although the microphone is considered the normal input sound source, you can also switch to the Line-In input, as you'll see when we discuss the SBMixer program.

After clicking the Record button, you'll see the status "Recording" and the maximum amount of time that you can record. This time depends on how much memory is available. If you need more time, try closing any windows that are running in the background.

In the future, you may even want to consider upgrading your system with extra RAM. In any case, the longest recording session allowed by Sound Recorder is 60 seconds.

If you don't overdo it, you should normally have enough memory available to handle your recording needs. However, you may be surprised by how fast the digital sound data accumulates. For example, at a sampling rate of 22,050 Hertz, 8-bit mono, the Sound Blaster Pro fills your WAV file with 22,050 bytes every second.

During recording, you can watch the resulting waveform of the digitized data in Sound Recorder's wave box.

Sampling with Sound Recorder

Sound editing and effects

Several functions are available to help you edit your sound files. A favorite is the **Echo** item under **Effects**.

Other functions let you adjust the volume and speed of a sound, and even reverse it.

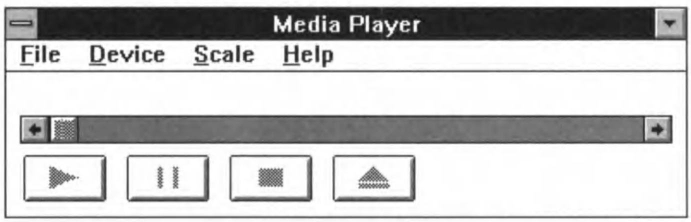
HINT

You'll probably want to test each of the different effects on your new sound file. However, remember that some cannot be undone. For example, the Echo effect cannot be removed. The same rule applies here as in Voice Editor under DOS: *Save first, then edit.*

3.4 Media Player

Media Player

Even if you don't want to record your own digital sound, another program from the Accessories group still should be interesting. The Media Player plays WAV files, similar to the Sound Recorder, but also provides additional input options, including MIDI files.



Media Player plays both WAV and MIDI files

If your system includes a CD-ROM drive and you've installed the required driver, Media Player even uses your drive as a CD player to play audio CD's.

As you select different input media, it's useful to select the appropriate position scale accordingly. The two **Scale** menu items are **Time** and **Tracks**.

The **Time** menu item is used in playing WAV and MIDI files and works the same as in Sound Recorder. The position scroll bar shows time intervals within the sound recording.

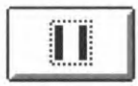
If you're playing a CD, the **Tracks** menu item will show the current position as a track number (i.e., the song selection that's currently playing).

Since you've just created a MIDI setup with MIDI Mapper, Media Player can play MIDI files on your system even without a MIDI

device. The Sound Blaster card assumes the role of a MIDI expander.

The Media Player buttons control the following functions:

The Play button starts the appropriate playing process for the device.



Play can be interrupted with the Pause button. Pressing Pause again resumes play at the same position where it was interrupted.

Play ends when you press the Stop button.

The Eject button is useful only for devices that have an automatic eject feature. This doesn't include the type of drive that uses a caddy. The Eject function cannot be used during play.

3.5 Sound Blaster Tools for Windows

In this section we'll discuss some useful Windows tools that are included with newer versions of the Sound Blaster card. If your card didn't include them, you should try to obtain them.

VOC2WAV and WAV2VOC

Converting VOC files to WAV format

The DOS programs VOC2WAV and WAV2VOC convert files from Creative Voice File format (VOC files) to Windows WAV format, and vice versa.

These programs are easy to use. For VOC2WAV, input the name of the VOC file you want to convert (the source file) as a command line parameter.

Following the source filename, you can specify the name the file should have after it has been converted (the target file). If you don't specify a target name, the source name is used. In either case, the target file receives the WAV extension instead of the VOC extension.

The filenames can be followed by additional conversion parameters, as described below.

Cx

This parameter tells the program whether the VOC file was recorded in mono or stereo mode. A "1" in place of the "x" indicates mono, and a "2" indicates stereo.

Rxx

This parameter specifies the desired sample rate of the output WAV file. The "xx" should be replaced with "11" for 11 KHz, "22" for 22 KHz, or "44" for 44 KHz. These are the only valid rates for WAV files.

Sxxx

This parameter specifies whether to unpack CT-Voice format silence blocks to their original extent. The value of "xxx" can be "ON" to restore the full silence period, or "OFF" to ignore it. The WAV format doesn't recognize silence blocks. The default is "OFF".

Lxxx

This parameter specifies whether to unpack repeat loops from the VOC file into the WAV file; this is another feature of VOC files that's not recognized by the WAV format. The value of "xxx" can be "ON" or "OFF". The default is "OFF".

If "ON" is specified, a loop in the VOC file that must be repeated three times will be copied three times into the WAV file. So a small VOC file containing many repeat loops can become quite large when converted.

Also, the sampling rates of VOC files are more flexible than those of WAV files. The file to be converted should have been recorded only at 11, 22, or 44 KHz. Different sample rates within a file will be ignored.

Packed data blocks are also not convertible and will be ignored by VOC2WAV.

WAV2VOC is even easier to run. You must specify the name of the file you want to convert to VOC format. The program handles everything else for you.

These two programs allow you to convert your favorite sounds from one format to the other, so they are accessible from both DOS and Windows.

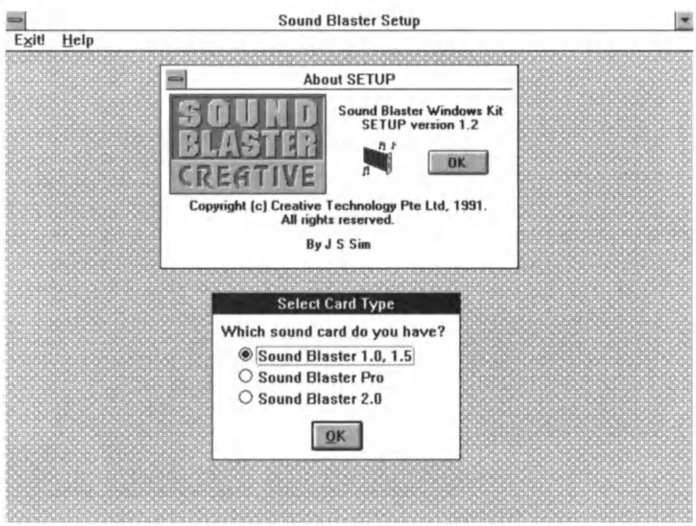
The Sound Blaster DLL

If you own Windows 3.0 without the MultiMedia Extension, Creative Labs provides some tools that enable you to use Sound Blaster under Windows.

To perform the necessary installation, first move to the \SBPRO\WINDOWS\ or \SB\WINDOWS\ directory. Then copy the file SNDBLST.DLL to the Windows main directory. Then make a new group using Program Manager and move all the Sound Blaster tools into this group.

To make the necessary changes to the WIN.INI file, start the Setup program by clicking the icon containing the expansion card.

You must specify which Sound Blaster card you've installed.



Starting the Setup program

Then enter the port address, interrupt number, and DMA channel of your configuration. These parameters will be stored in a

[SoundBlaster] section of WIN.INI. With a standard installation this looks as follows:

```
[SoundBlaster]
Port=220
Int=7
DMA=1
```

Now you can use the other tools even without Windows 3.1 or the MultiMedia Extension. Occasionally these tools conflict with the sound management of Windows 3.1, possibly even resulting in a system crash.

HINT

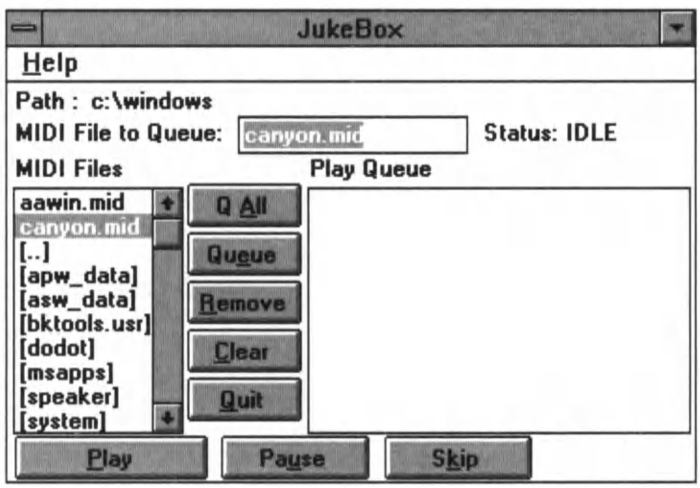
If you won't be using the Jukebox program, which we'll describe next, you don't have to install the DLL. You shouldn't use the Windows 3.0 tools from Creative Labs with Windows 3.1.

Jukebox

Plays your MIDI files under Windows

Jukebox plays your MIDI files under Windows. To do this, select these files and place them in a queue. From there, Jukebox plays them one after another.

By using Jukebox, you can put together a nice selection of background music for long Windows sessions.



Jukebox window

Most functions of the Jukebox program are taken over by Media Player under Windows 3.1. Unfortunately, Media Player doesn't

offer Jukebox's queue management versatility. In Jukebox, you can pause, clear the queue, or skip to the next song.

Sound Blaster Pro Mixer

Most useful Sound Blaster utility available

The Sound Blaster Pro Mixer is the most useful Sound Blaster utility available. It can be used only with Sound Blaster Pro, which is the only card that supports software mixing of all audio sources. Like WAV2VOC.EXE and VOC2WAV.EXE, the Sound Blaster Pro Mixer (SBP_MIX.EXE) is accessible from DOS.

With Sound Blaster Pro Mixer, you can set all volume parameters of SB Pro. These sources are listed int the following table:

Source	Function
Mas	Master volume
FM	Internal FM voices
CD	Audio CD in a CD-ROM drive
Mic	Microphone
Lin	Line-In input
Voc	Digitized sound channel (also called voice)

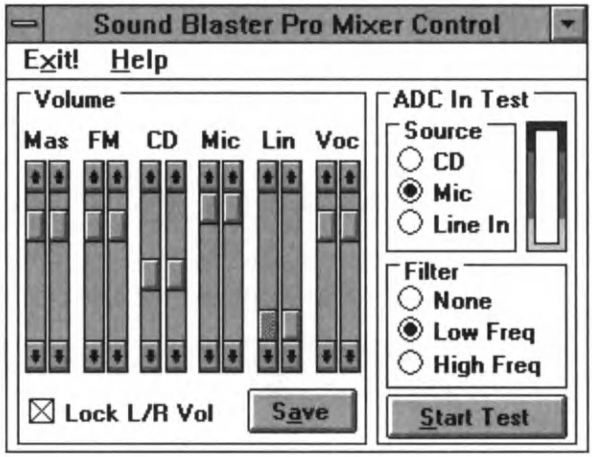
The following displays the Sound Blaster Pro Mixer.

The Sound Blaster Pro Mixer

ADC source (analog source to be digitized) can be selected, as well as a low-pass or high-pass filter.

If you have the new Sound Blaster Pro drivers for Windows 3.1, you should've also received a new version of Sound Blaster Pro

Mixer. This version also allows the stereo channels to be controlled individually.



Sound Blaster Pro Mixer for Windows

A colored gauge next to the ADC source selection box lets you test the input signal strength of the ADC source. It appears when you click the [Start Test] button.

You can also switch off the filter completely and save the set values.

The Mixer enables you to play background music from a radio, tape recorder, or CD over your Sound Blaster Pro during a Windows session.

Source selection

Simply connect your radio, tape recorder, or CD player to the Line-In input of SB Pro and switch it on. Then use Sound Blaster Pro Mixer to adjust the master volume and Line-In volume to select the desired volume.

One of Sound Blaster Pro Mixer's most useful features is its function as a source selector for analog-to-digital conversion.

The microphone is the sole input source for sound to be digitized on the Sound Blaster 1.0 and 1.5. Sound Blaster Pro has CD-ROM and Line-In sources as well.

Sound Recorder cannot recognize the different sources. But by first selecting the desired source in Sound Blaster Pro Mixer, you can

then use Sound Recorder to record digitized sound from any of the three analog sources.

If you frequently use your Sound Blaster card under Windows, you may want to place Sound Blaster Pro Mixer in the Autostart group.

As with Jukebox, the Windows 3.0 (mono) version of Sound Blaster Pro Mixer shouldn't be run under Windows 3.1, because problems can occur.

3.6 MIDI under Windows

NOTE

In this section we'll discuss how MIDI relates to Windows. If you're not familiar with the basic concepts of the MIDI standard yet, it may be helpful to review Chapter 4 before reading this section.

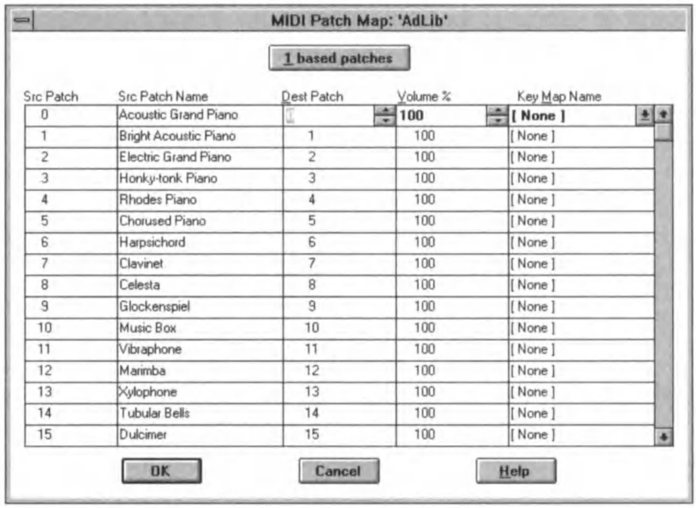
The multimedia functions of Windows 3.1 provide device-independent MIDI support. This enables you to reference your Sound Blaster card or any other MIDI device with full compatibility.

Almost everything in MIDI is specified exactly. The one exception is the numbering of the various instruments and their sounds. This is only loosely incorporated in General MIDI specifications. There isn't a clear standard because of the various synthesizers produced by different manufacturers.

Consequently, a MIDI song written for one synthesizer can sound quite different when played on another one. Horns can replace strings, for example, producing a completely different effect than the composer intended.

Patch Mapper

Windows 3.1 has a patch mapping capability within its MIDI Mapper to solve this problem. You can use this to accommodate the instrument voices of a General MIDI file to any synthesizer. In this instance a patch is simply a particular instrument sound.



Src Patch	Src Patch Name	Dest Patch	Volume %	Key Map Name
0	Acoustic Grand Piano		100	[None]
1	Bright Acoustic Piano	1	100	[None]
2	Electric Grand Piano	2	100	[None]
3	Honky-tonk Piano	3	100	[None]
4	Rhodes Piano	4	100	[None]
5	Chorused Piano	5	100	[None]
6	Harpsichord	6	100	[None]
7	Clavinet	7	100	[None]
8	Celesta	8	100	[None]
9	Glockenspiel	9	100	[None]
10	Music Box	10	100	[None]
11	Vibraphone	11	100	[None]
12	Marimba	12	100	[None]
13	Xylophone	13	100	[None]
14	Tubular Bells	14	100	[None]
15	Dulcimer	15	100	[None]

Patch Mapper in Windows 3.1

Under "Src Patch" are the sounds that would be sent by a General MIDI file, numbered in sequence from 0 through 127.

Click the button marked [1 based patches] if your synthesizer recognizes patch numbers from 1 to 128 instead of 0 through 127.

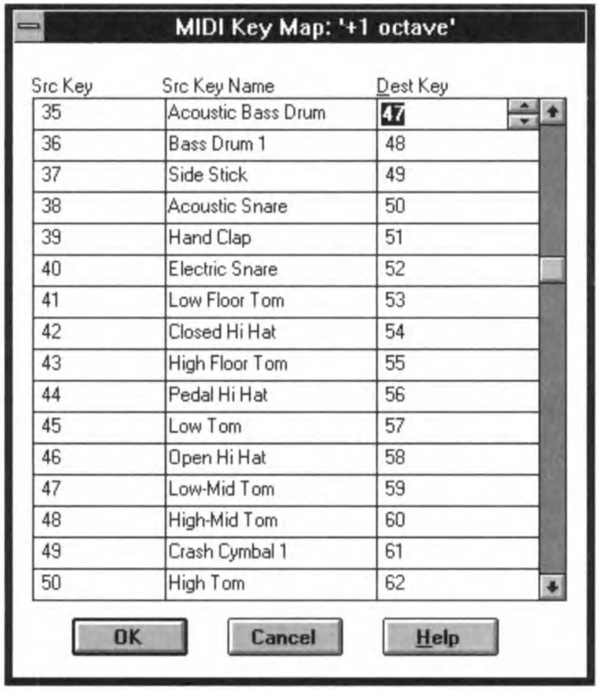
The source patch name describes the normal sound of the source patch. Source patch 0, for example, is "Acoustic Grand Piano". If your synthesizer has this sound at a different position, for example 42, you must place 42 in the "Dest Patch" column for the row documenting this source.

The "Volume %" column specifies the volume at which the destination patch should be played.

The "Key Map Name" column is needed if your synthesizer plays keys in a different registration than that defined by General MIDI. If it is an octave too low, for example, you will need a key map of "+1 octave". This is important if you're creating a patch map for percussion sounds and your synthesizer plays a bass drum instead of a hi-hat.

Basic percussion sounds on most synthesizers are on a separate instrument voice, with certain keys, or pitches, assigned to certain sounds. A C-sharp may produce an open hi-hat and an F-sharp a snare drum. If the synthesizer's pitch doesn't match the General MIDI standard exactly, the wrong drum sound is heard.

If this occurs, you can use MIDI Mapper to make another type of translation table, called the key map, which is then inserted into the patch map.


MIDI Key Map: '+1 octave'

Src Key	Src Key Name	Dest Key
35	Acoustic Bass Drum	47
36	Bass Drum 1	48
37	Side Stick	49
38	Acoustic Snare	50
39	Hand Clap	51
40	Electric Snare	52
41	Low Floor Tom	53
42	Closed Hi Hat	54
43	High Floor Tom	55
44	Pedal Hi Hat	56
45	Low Tom	57
46	Open Hi Hat	58
47	Low-Mid Tom	59
48	High-Mid Tom	60
49	Crash Cymbal 1	61
50	High Tom	62

Example of a percussion key map

If your Windows system is properly configured in MIDI Mapper, MIDI files will sound good on your synthesizer even if it doesn't conform to all the General MIDI specifications.

Windows can even handle both base-level synthesizers and extended-level synthesizers.

Base-level

Base-level synthesizers are devices that meet only the current minimum MIDI requirements. They can play at least six notes with three different melodic voices and three different percussion voices simultaneously.

These synthesizers receive their data on channels 13 through 16. Channels 13 through 15 are the melodic channels and channel 16 is the percussive channel.

Extended-level

Extended-level synthesizers are devices that are capable of meeting the full spectrum of MIDI performance requirements. They

must be able to play 16 notes with 9 melodic voices and 16 notes with 8 percussive voices simultaneously.

They receive their data on MIDI channels 1 through 10, with channels 1 through 9 being used for the melodic voices and channel 10 for the percussion.

Basic MIDI, Extended MIDI

Therefore, Windows has two different channel assignments, one for Basic MIDI and one for Extended MIDI.

A Windows MIDI file must have control information for both levels of synthesizers.

MIDI Mapper gives your hardware the appropriate data when outputting a MIDI file. So if you have a base-level synthesizer connected (like the Sound Blaster card itself), extended-level data is suppressed and vice versa.

If you want to play a MIDI file that doesn't conform to the Microsoft MIDI standard and, therefore, doesn't contain data for both types of synthesizers, you can create a custom setup in MIDI Mapper. This enables you to remap the percussion channel from 10 to 16, for example, or vice versa.

If your synthesizer is overloaded by more notes than it can handle simultaneously, the driver handles this situation by letting the first sounds fade away.

Overall, Windows manages MIDI requirements very well. We can expect more MIDI applications to be developed in the future.

To help you create a patch map for your synthesizer, the following table shows the Microsoft standard. The individual sounds are classified by instrument type to help you locate the desired sound.

Piano Sounds			
000	Acoustic Grand Piano	001	Bright Acoustic Piano
002	Electric Grand Piano	003	Honky-Tonk Piano
004	Rhodes Piano	005	Chorused Piano
006	Harpsichord	007	Clavinet

Chromatic Percussion			
008	Celesta	009	Glockenspiel
010	Music Box	011	Vibraphone
012	Marimba	013	Xylophone
014	Tubular Bells	015	Dulcimer

Organ Sounds			
016	Hammond Organ	017	Percussive Organ
018	Rock Organ	019	Church Organ
020	Reed Organ	021	Accordion
022	Harmonica	023	Tango Accordion

Guitar Sounds			
024	Acoustic Guitar (nylon)	025	Acoustic Guitar (steel)
026	Electric Guitar (jazz)	027	Electric Guitar (clean)
028	Electric Guitar (muted)	029	Overdriven Guitar
030	Distortion Guitar	031	Guitar Harmonics

Bass Sounds			
032	Acoustic Bass	033	Electric Bass (fingered)
034	Electric Bass (picked)	035	Fretless Bass
036	Slap Bass 1	037	Slap Bass 2
038	Synth Bass 1	039	Synth Bass 2

String Sounds			
040	Violin	041	Viola
042	Cello	043	Contrabass
044	Tremolo Strings	045	Pizzicato Strings
046	Orchestral Harp	047	Timpani

Ensemble Sounds			
048	String Ensemble 1	049	String Ensemble 2
050	Synth Strings 1	051	Synth Strings 2
052	Choir Ahs	053	Voice Oohs
054	Synth Voice	055	Orchestra Hit

Brass Sounds			
056	Trumpet	057	Trombone
058	Tuba	059	Muted Trumpet
060	French Horn	061	Brass Section
062	Synth Brass 1	063	Synth Brass 2

Reed Instruments			
064	Soprano Sax	065	Alto Sax
066	Tenor Sax	067	Baritone Sax
068	Oboe	069	English Horn
070	Bassoon	071	Clarinet

Flute Wind instruments			
072	Piccolo	073	Flute
074	Recorder	075	Pan Flute
076	Blown Bottle	077	Shakuhachi
078	Whistle	079	Ocarina

Synth Lead Sounds			
080	Lead 1 (square)	081	Lead 2 (sawtooth)
082	Lead 3 (calliope)	083	Lead 4 (chiff)
084	Lead 5 (charang)	085	Lead 6 (voice)
086	Lead 7 (fifths)	087	Lead 8 (bass + lead)

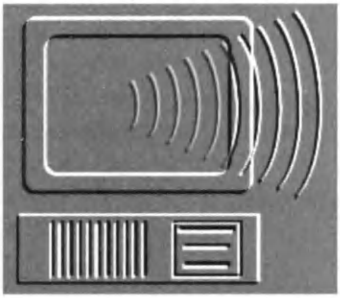
Synth Pad Sounds			
088	Pad 1 (new age)	089	Pad 2 (warm)
090	Pad 3 (polysynth)	091	Pad 4 (choir)
092	Pad 5 (bowed)	093	Pad 6 (metallic)
094	Pad 7 Halo Pad	095	Pad 8 Sweep Pad

Synth Effect Sounds			
096	SFX 1 (rain)	097	SFX 2 (soundtrack)
098	SFX 3 (crystal)	099	SFX 4 (atmosphere)
100	SFX 5 (brightness)	101	SFX 6 (goblins)
102	SFX 7 (echoes)	103	SFX 8 (sci-fi)

Folk Instruments			
104	Sitar	105	Banjo
106	Shamisen	107	Koto
108	Kalimba	109	Bagpipe
110	Fiddle	111	Shanai

Percussive Instruments			
112	Tinkle Bell	113	Agogo
114	Steel Drums	115	Woodblock
116	Taiko Drum	117	Melodic Tom
118	Synth Drum	119	Reverse Cymbal

Effects			
120	Guitar Fret Noise	121	Breath Noise
122	Seashore	123	Bird Tweet
124	Telephone Ring	125	Helicopter
126	Applause	127	Gunshot



Chapter 4

MIDI and Sound Blaster

MIDI and Sound Blaster

With Sound Blaster you can experience the world of computer-generated music without purchasing expensive hardware. MIDI makes this possible.

In this chapter we'll discuss MIDI in detail. You'll learn what MIDI can do and how you can use it with your Sound Blaster card.

Since MIDI is a very complex subject, involving various types of hardware and software, we can provide only a general overview of MIDI.

With this in mind, we use the general term "keyboard" to describe a MIDI instrument, although many other types of MIDI instruments exist (guitars, wind instruments, percussion).

4.1 A Quick Look at MIDI

What is MIDI?

MIDI is an acronym for "Musical Instrument Digital Interface". It acts as an interface between musical instruments and a computer.

MIDI beginnings

Compared to the history of music, MIDI is a recent development. In the 1970s synthesizer technology was mainly analog because digital technology was less-developed and more expensive.

At that time, networking analog synthesizers required advanced skills in electronics, making such networking a difficult task.

The first digital musical instruments that appeared on the market weren't well-suited for an amateur musician. They were mostly closed systems that either had no outside interface at all, or were compatible only with other devices from the same manufacturer.

The importance of a standard

A standard, which would enable different keyboards to interface with each other, was needed. Also, an acceptable standard would

support the addition of other electronic instruments, effects devices, drum machines, and various studio accessories. Finally, this standard would need to convey relevant data other than musical notes.

Developing this type of standard was a difficult task in terms of technology. However, in addition to the technical aspects, this task seemed almost impossible because it involved cooperation from numerous competing manufacturers of electronic musical instruments. Ultimately, only Kawai, Korg, Roland, Sequential Circuits, and Yamaha supported the agreement.

In 1983, Sequential Circuits and Roland introduced the first keyboards with the standard MIDI interface. Although the capabilities of this interface were still very limited, it was the basis for further developments in this area.

IMA

MIDI Specification 1.0 was established. To ensure that all manufacturers complied with the standard, the International MIDI Association (IMA) was formed in the United States.

Although the beginning was difficult and problems still exist, today all manufacturers adhere to the MIDI standard.

Various weaknesses in the MIDI system prompted some manufacturers to abandon the standard briefly. However, most of them lost market share because of this decision and eventually returned to supporting the standard. This indicated that MIDI had been accepted by the users, who are primarily musicians. This acceptance is easy to understand because MIDI's advantages far exceed its disadvantages.

Add devices easily

An important advantage of the MIDI system is that you can easily connect devices. Before MIDI, analog devices had to be wired together through solder, with hardware modifications to compensate for different voltages. Today, you can run a MIDI cable from the MIDI OUT port of a master keyboard to the MIDI IN port of a slave keyboard. When you switch on both instruments, playing a note on the master keyboard plays the same note on the slave keyboard.

In this way, you can connect a (theoretically) unlimited number of slave instruments to the master keyboard.

Various MIDI manufacturers

Another advantage of MIDI is its compatibility standard. This enables you to use keyboards and other devices from various

manufacturers. MIDI hardware can be removed, added, and exchanged whenever desired. Your system continues playing, in perfect harmony, with every new device.

MIDI understands all types of data

The types of data handled by a MIDI system are as varied as the sequencers, drum machines, expanders, mixers, and other devices they control. They include information about rhythm, pitch, attack dynamics, and the variety of sounds themselves.

All this information passes through the MIDI system and must be recognized and processed as needed by each device.

MIDI and computers

The most important advantage of MIDI is its ability to integrate a computer into the musical system. The use of a computer brings an exciting new dimension to sound.

You can save your songs directly to the computer and call them back at any time. Without even knowing how to read notes, you can produce your songs in proper musical notation and share these songs with others. It's also possible to save a favorite sound, which you've created with your synthesizer, on your computer's hard disk.

We've mentioned only a few of MIDI's many advantages. You'll discover many more as you continue to learn about MIDI.

Today MIDI is the standard for constructing music systems that use various types of component devices.

4.2 Basic Concepts of MIDI

MIDI terms

In this section, we'll briefly describe the most important concepts of MIDI. This should provide an overview of the basic topics and terms.

Master device

A master device is one of the most important components of a MIDI system. All other connected devices are controlled by this device. The most common master devices are keyboards or computers.

Keyboard controllers

Keyboard controllers are high-quality keyboards that can send MIDI messages but seldom contain their own system of tone generation. Their main purpose is controlling a MIDI system.

Keyboard synthesizers

A keyboard synthesizer includes both a keyboard and a tone generator. The general term "keyboard" usually implies a keyboard synthesizer. A good keyboard synthesizer can also be used as the master keyboard.

Slave devices

This term encompasses all MIDI system components controlled by the master device. So these devices are dependent on the master device. Expanders, effects devices, and drum machines are typical slave devices.

Expanders

An expander is a synthesizer without a keyboard. It must be controlled by a master device. Since it contains fewer moving parts, an expander often costs less than its keyboard equivalent. Also, in large MIDI systems it may be more practical to use expanders because they occupy less physical space than a keyboard synthesizer.

Sequencers

A sequencer is a device or software product on which you can record an entire musical composition (sequence) for playback. On many sequencers, you can record individual channels while playing back existing channels. This is similar to the overdub capability found on reel to reel tape recorders. The entire sequence can then be played back at the press of a key.

Drum machines

These devices, also called drum computers, have replaced true percussion instruments in many studios. A drum machine acts as a rhythm generator, producing sounds of various drum types and other percussive instruments. Like melodic instrument voices, these sounds can also be sent over MIDI.

Velocity

Velocity refers to the manner and speed with which the key, drum pad, or string producing a sound is activated. In other words, the velocity (speed) at which the player attacks a note defines the volume of the note. Most MIDI keyboards are velocity-sensitive, which means that the faster (harder) the user plays a note, the louder that note will sound.

The MIDI system provides 128 levels for velocity. This is slightly limited when compared to the much finer nuances achieved with acoustical instruments.

Aftertouch

Aftertouch refers to the degree of pressure exerted on a key during play, until it is released. There are two types of aftertouch: Monophonic and polyphonic. A keyboard with monophonic aftertouch senses only the hardest pressed key, while an instrument with polyphonic aftertouch can sense the pressure on each key individually when a chord is played. This latter feature is found only on more expensive keyboards.

MIDI measures both forms of aftertouch in 128 increments.

Pitch bend

A good keyboard should be able to bend pitches the way a horn or guitar does as it slides up or down into a note. This effect is accomplished using a wheel to control the pitch. There is also a MIDI message to control this effect.

The MIDI channels

You can do a lot more with MIDI than simply play music on a single channel. A good MIDI system is extremely versatile and powerful. The Sound Blaster card offers at least 11 separate channels for FM voices. This means that you can simultaneously play 11 different sounds, each at its own pitch and for its own duration, to produce dramatic musical works.

A MIDI device must be able to recognize and process its own data from among all the information passing through the network of MIDI cables. Therefore, the system is divided into channels.

When the master device sends data, it also supplies a channel number for the device that's supposed to process it. A device that is set for this particular channel will process the data, while all others will ignore it. So with the right channel number you can address the right device.

It's very interesting when a master device can send data to several channels at once.

Channel Mode messages

Channel Mode messages are used to place the devices connected to a certain channel into a certain operating mode.

Omni mode

Following the OMNI ON message, all slave devices respond to MIDI messages, regardless of channel number. OMNI OFF reactivates channel-specific addressing.

Poly mode

When a POLY ON message is sent to a slave device, the device can be played polyphonically. This means that, depending on the number of voices for which it's designed, the device can play multiple voices simultaneously.

Many devices can also divide their sounds by pitch so that on such devices more than one sound can be controlled over a single channel.

Mono mode

In a slave device's mono mode, not all of its voices are assigned to a common channel. Instead, each voice can be assigned to its own channel. The advantage of this process is that you can send each voice its individual data about the desired sound of the tone to be played. The following table shows an example of the channel voice assignments:

MIDI channel	Sound	Number of voices
1	Piano	mono
2	Bass	mono
3	Strings	mono
4	Horns	mono
...	...	...
8	Choir	mono

However, each device can only play one voice at a time. A partial solution would be to lay out multiple channels for one voice. The following is an example (the numbers in parentheses here represent each voice):

MIDI channel	Sound	Number of voices
1	Piano (1)	mono
2	Piano (2)	mono
3	Piano (3)	mono
4	Piano (4)	mono
5	Bass	mono

MIDI channel	Sound	Number of voices
6	String (1)	mono
...	...	...
16	Choir (4)	mono

To provide a better solution, mono mode was further developed and is now multi mode. In this mode a synthesizer can be referenced both polyphonically and on multiple channels. The following table illustrates a multi mode arrangement:

MIDI channel	Sound	Number of voices
1	Piano	3
7	Bass	mono
9	Choir	4

Omni, poly, and mono modes can be combined as desired, except that the poly and mono modes are mutually exclusive. Valid combinations, then, are OMNI ON/POLY ON, OMNI ON/MONO ON, OMNI OFF/POLY ON, and OMNI OFF/MONO ON.

Local mode

Another channel mode message is LOCAL ON/LOCAL OFF. With a keyboard synthesizer, you have both a keyboard and a tone generator. LOCAL OFF disconnects these components from each other. What is played on the keys is translated into MIDI messages and sent to other parts of the system, bypassing the tone generator.

So you could use the keyboard as a master keyboard over MIDI OUT, while simultaneously driving the tone generator with a sequencer over MIDI IN. The sequencer could be playing something completely different than what you're playing on the keys.

Sending a LOCAL ON message returns the keyboard to normal, so it controls its own tone generator.

Switching all notes off

Another useful mode message is the ALL NOTES OFF message. A mode change with this message stops all active tones on the slave device. A garbled data signal can sometimes result in a note continuing to sound after it should have been switched off. In such

a case, you must either find the note in question and switch it off, or send an ALL NOTES OFF message.

Channel Voice messages

Channel Voice messages relate to a single voice, and therefore to a single note that must be played on a certain channel.

NOTE ON/NOTE OFF

NOTE ON and NOTE OFF are the most fundamental messages affecting the playing of a sound. These messages determine the desired channel number, the pitch, and note velocity.

AFTERTOUCH and PITCH BEND

Other messages available to control an individual tone are AFTERTOUCH and PITCH BEND. These terms were explained earlier.

PROGRAM CHANGE

A PROGRAM CHANGE message changes the patches (sounds) on the devices connected to a particular channel. There are 128 different program numbers possible.

The number of actual patches varies with the synthesizer. For example, a Casio CZ series synthesizer has 48 possible sounds, while the Kawai K1 has up to 128 possible sounds available. It's up to you to send a meaningful program number to each device, since not all devices are capable of producing all 128 sounds. Also, you must ensure that the slave device allows a MIDI-controlled program change. This property is set with the PROGRAM CHANGE ON or PROGRAM CHANGE OFF message.

Right program, wrong sound?

Remember that the program number you send to a slave device with the PROGRAM CHANGE message probably doesn't match the master device's own number for that sound. Your master device may have a drum sound stored at patch 76, but you may have to send data for this sound as patch 100 to the slave device.

This is because a synthesizer's sounds are usually grouped thematically, with non-standard groupings that vary from device to device. Find the MIDI patch listing in your synthesizer manual to determine which sound corresponds to which MIDI program number.

CONTROL CHANGE

The most comprehensive channel voice message is the CONTROL CHANGE message. This message encompasses a wide variety of control features for different devices.

The way the pedals of a piano affect the piano's sound is a good example of how this message works. This quality is indispensable

in professional music and must be able to be encoded and reproduced by a MIDI system.

The CONTROL CHANGE message was developed to accommodate such types of sound control, some of which are extremely subtle. It allows 128 different slave device parameters to be addressed. The parameters themselves aren't fully defined in the General MIDI standard. Only numbers 0 through 31 are reserved for specific controllers. Numbers 64 through 95 refer to pedals and other switches, and 122 through 127 refer to the Channel Mode messages previously described. Numbers 32 through 63 and 96 through 121 are undefined.

Which controller number controls which parameter depends on the manufacturer, except in the case of those specifically defined by General MIDI. This is because, when the standard was written, all types of control capabilities that would be developed in the future couldn't be predicted.

Some controllers can address more than the 128 different values available for most other MIDI messages. These use a higher-resolution mode, in which two data bytes accompany the message instead of one. The number of attainable levels is then 16384.

System Common messages

Unlike channel messages, System Common messages are intended for the entire system, instead of for only individual channels.

System Common messages synchronize all devices within a system, so they will play smoothly together.

TUNE REQUEST

The TUNE REQUEST message causes all devices relating to a specific function to retune themselves internally. This means that, within a given device, all voices are adjusted among themselves. This works only at the device level. You can't tune two devices to each other using this message (you still have to tune by ear).

SONG SELECT

The SONG SELECT message lets you activate certain songs that are stored by number in a sequencer or drum machine. Up to 128 songs are possible under MIDI.

SONG POSITION POINTER

With the SONG POSITION POINTER message you can tell a sequencer exactly where to start or continue playing within a song. This can be done to synchronize the sequencer with a drum machine, for example.

Position is counted in sixteenth beats from the beginning of the sequence. The message has two data bytes, allowing a maximum position of 16384 sixteenth beats or 1024 beats from the start.

System Exclusive messages

System Exclusive messages are device-specific messages that are meant to be processed by only a particular brand of device. You can use them to exchange sounds between similar synthesizers or store the sounds on a data carrier.

The structure of this message is very flexible. The one fixed portion is the status byte, which indicates that a System Exclusive message follows. After the status byte is a manufacturer identifier that indicates the appropriate devices.

The rest of the message depends on the manufacturer and the task to be performed. At the end of this data is another special status byte indicating the end of the message.

Consult your device manual for additional information on System Exclusive messages for your device. Many message types and formats are possible.

System Real-Time messages

System Real-Time messages control the timing of the system and ensure that all components play together as a unit.

TIMING CLOCK and START messages

The TIMING CLOCK message counts time for the system and enables the components to keep in sync.

The START message causes a device, such as a sequencer or drum machine, to play a specified song from the beginning.

STOP message

The STOP message causes the sequencer or drum machine to stop playing and wait.

CONTINUE message

The CONTINUE message causes the device to continue playing at the place where it had stopped. This proceeds in sync with all other devices.

SYSTEM RESET message

The SYSTEM RESET message sets all devices in a MIDI system back to their initial status. It works essentially like rebooting a computer and, as with a computer, is useful for recovering from certain errors.

ACTIVE SENSING message

This type of message is constantly sent out by a master device when no other data is being transmitted. If a slave device doesn't receive data or an ACTIVE SENSING message, a broken connection is indicated. The slave device will switch off all active voices when this occurs.

This feature is necessary because a note usually continues playing until its NOTE OFF message is received.

4.3 Getting Started with MIDI

Now that you know the basic concepts of MIDI, we'll show you how to create a simple MIDI system. This should help you overcome any problems you may encounter when trying to create MIDI music. Then the only obstacle you'll encounter will probably be the high cost of many MIDI devices.

The system described in this section is a very modest one, which includes only a MIDI-controlled keyboard.

4.3.1 MIDI cables

Since all MIDI experiments begin with connecting devices, we'll start by discussing the actual connecting cables.

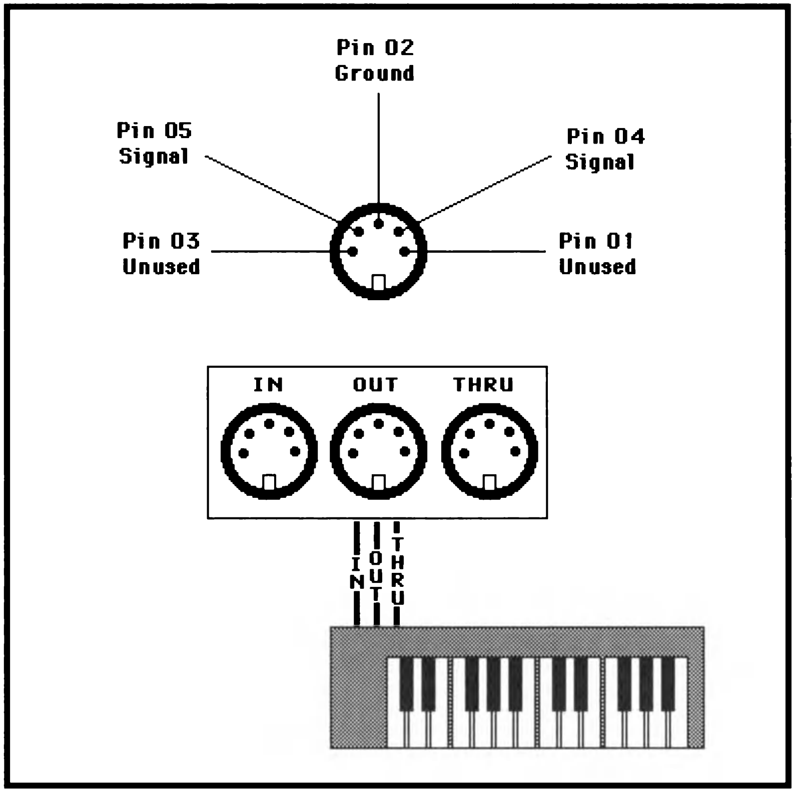
The MIDI cable

Similar to most computer mouse devices, MIDI uses the serial method of data transfer. So MIDI cables are two-poled. Cables should be shielded and shouldn't be longer than 32 to 50 feet (10 to 15 meters). However, the cables should be as short as possible because quality decreases as the length increases.

MIDI connector

The MIDI specification also includes connectors. There are two types of connectors.

The type most commonly used is the 5-pin 180-degree DIN plug and jack. Almost all MIDI devices are equipped with the DIN style plug. The reason for this is probably because they are cheaper than the 3-pin XLR connection. The use of this connector is a disadvantage of MIDI systems, because it's more susceptible to damage, which affects the signal.



The 5-pin DIN plug with MIDI wiring

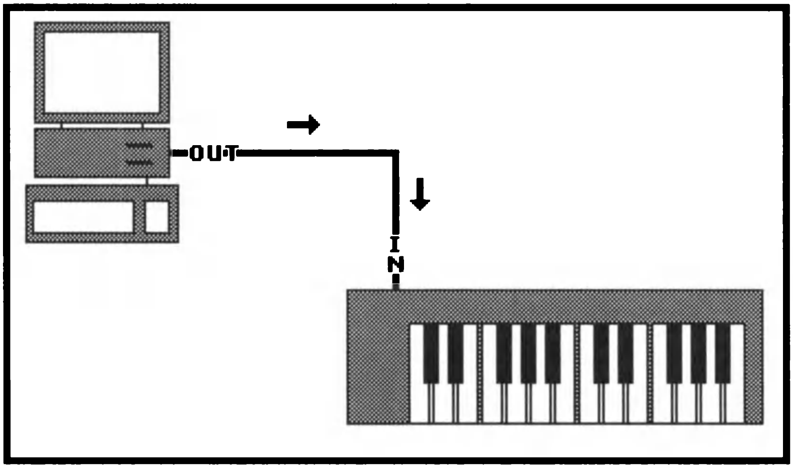
4.3.2 Building a MIDI system

Hooking up the MIDI keyboard

Connecting a MIDI keyboard is easy. Simply connect the MIDI OUT port of the Sound Blaster card to the MIDI IN port of the keyboard.

This forms the simplest possible MIDI system, with your computer and Sound Blaster card acting as master device, and the keyboard acting as slave.

The MIDI keyboard can now be driven by your computer using MIDI data.



A simple MIDI system

If you want to extend your system further, there are two ways to connect additional devices.

Daisy chaining

Daisy chaining requires connecting keyboards in series. To connect devices in a daisy chain formation, all but the first and last devices in the chain must have the following:

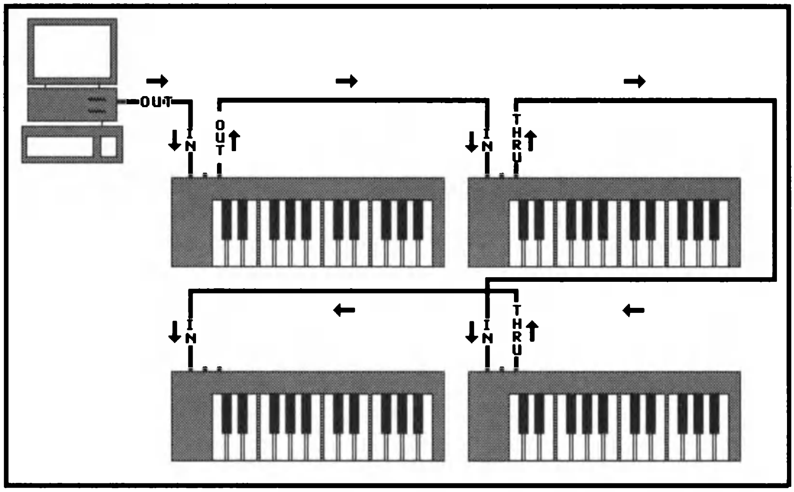
- MIDI IN port
- MIDI OUT port
- An additional port called MIDI THRU

Many keyboards have a MIDI THRU port, although it isn't required by the general MIDI specifications.

MIDI THRU

Data received on the MIDI IN port of a chained keyboard not only goes to the keyboard's microprocessor, but is also simultaneously sent to the MIDI THRU port and on to the next device.

In this arrangement, data can be sent from one device to the next across the length of the entire chain.



An example of daisy chaining

Daisy chaining is a convenient way of constructing a MIDI system. However, if the daisy chain gets too long, problems can occur.

Linking too many devices in a single daisy chain leads to two kinds of problems. First, the data signal deteriorates slightly as it passes through a MIDI device. The more devices in a chain, the more likely a signal corruption will occur. This results in unreadable data.

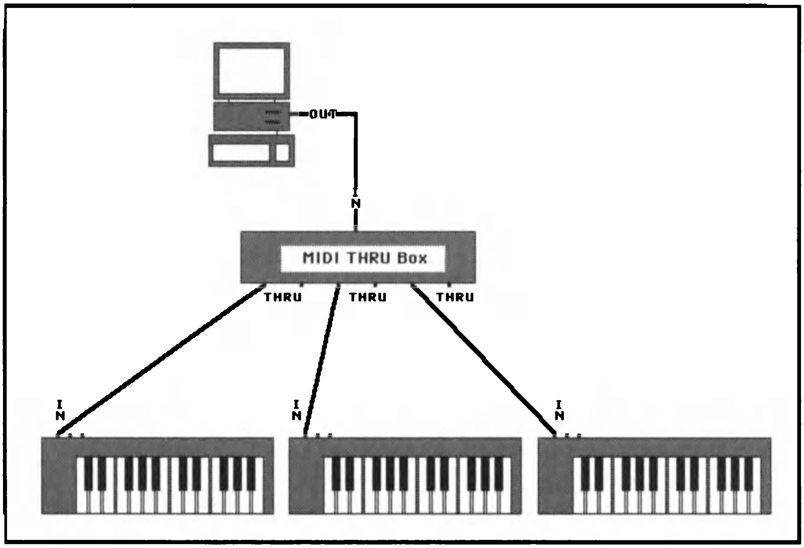
Second, the data requires time to traverse the daisy chain. Some MIDI THRU ports delay MIDI messages, and this delay increases for every device in the daisy chain. This interferes with the synchronization needs of a music system.

The limitations of this configuration can be tested with each setup. If necessary, there is another solution.

Star network

Although the star network requires some extra hardware, it offers three advantages over simple daisy chaining.

While the MIDI THRU port is needed in all but the last device of a daisy chained system, with the star network none of the devices require MIDI THRUs. Instead, each device in the system is connected to a special junction device called a MIDI THRU box.



An example of star networking

Another important advantage is that there is limited signal corruption. This is because the data must make its way only through the MIDI THRU box to get to any other device.

The third advantage of a star network is that there aren't any timing disturbances because data goes out from the box to all other devices simultaneously.

4.3.3 MIDI language

As you know, to communicate with your computer, you must use a special computer language. Which language you use depends on the task you want to perform. For example, you can write a program in assembly language or Pascal.

MIDI devices also have their own special language. If you'll be using existing MIDI software, you won't need to know every detail of every message.

However, if you'll be writing your own software for MIDI applications, or if you simply want to know what goes on inside a MIDI system, the following overview of MIDI messages should be useful.

With this information and the previous explanations of MIDI terms, you can begin writing even complex application programs. Even if you only want to work on a simple MIDI sequencer program, you should be familiar with the various messages.

The basic structure of the MIDI language is simple. There are only two types of bytes used in MIDI: Status bytes and data bytes.

Status bytes and data bytes

The most significant bit of a status byte is always 1. So a status byte is always of the form 1xxxxxxx. In a data byte the most significant bit is always 0, giving it the form 0xxxxxxx. A status byte is sent first, followed by one or two data bytes, depending on the message.

The first note

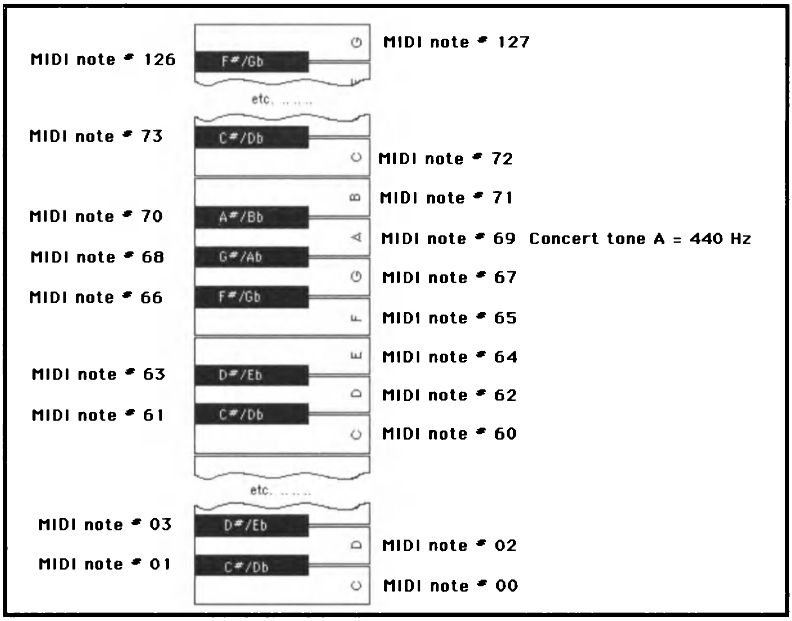
To play a note on a channel, two messages are needed. NOTE ON starts the note and NOTE OFF stops it. Both messages are Channel Voice messages.

NOTE ON

The status byte for NOTE ON is 1001xxxx, where xxxx is the binary number of the desired channel. So the status byte 10010000 sends a NOTE ON message to Channel 0 and 10011111 sends a NOTE ON message to Channel 15.

The data byte that follows describes the note to be played. There are 128 possible values, from 00000000 = 0 through 01111111 = 127. Note 01000101 = 69 is a standard A pitch with a frequency of 440 Hertz.

Other pitches can be derived from this. One note number higher goes a half step up, one lower goes a half step down.



MIDI notes on the keyboard (excerpt)

The second data byte of a NOTE ON message contains the VELOCITY data. This also has a value of 0 through 127.

NOTE OFF

A note stops playing with the NOTE OFF message. The status byte of NOTE OFF is 1000xxxx, where xxxx is again the channel number.

The first data byte again contains the note number and the second contains the decay dynamics.

With only these messages, you can already write your first simple MIDI music program.

The following is an overview of all the MIDI messages, with their status byte and data byte layouts.

Channel Voice messages

The various Channel Voice messages, with their corresponding data bytes, are constructed as follows. All bytes are expressed in binary notation.

NOTE OFF message	
Status byte	1000xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is the MIDI note from 0 through 127.
Data byte 02	0xxxxxxx, where xxxxxxx is the VELOCITY value from 0 through 127.

NOTE ON message	
Status byte	1001xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is a MIDI note from 0 through 127.
Data byte 02	0xxxxxxx, where xxxxxxx is the VELOCITY value from 0 through 127. Keyboards without velocity-sensitive keys default to 64.

POLYPHONIC AFTERTOUCH message	
Status byte	1010xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is a MIDI note from 0 through 127.
Data byte 02	0xxxxxxx, where xxxxxxx is an AFTERTOUCH value from 0 through 127.

CONTROL CHANGE message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is a controller number from 0 through 127. The values 122 through 127 are reserved for Channel Mode messages.
Data byte 02	0xxxxxxx, where xxxxxxx is a controller value from 0 through 127.

Some CONTROL CHANGE numbers are defined by MIDI specifications, as the following table shows:

Number	Parameter	Value
1	Modulation wheel	0-127
2	Breath controller	0-127
3	Undefined	
4	Foot controller	0-127
5	Portamento time	0-127
6	Data entry MSB	0-127
7	Main volume	0-127
64	Sustain	0,127
65	Portamento	0,127
66	Sostenuto pedal	0,127
67	Soft pedal	0,127
68	Data increment	127
69	Data decrement	127

General specification

000 - 031	Controller 0-31 high byte
032 - 063	Controller 0-31 low byte
064 - 095	Switches, pedals
096 - 121	Undefined

PROGRAM CHANGE message	
Status byte	1100xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is a program number from 0 through 127, representing a sound patch.

AFTERTOUCH message	
Status byte	1101xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is an AFTERTOUCH value from 0 through 127.

PITCH BEND message	
Status byte	1110xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	0xxxxxxx, where xxxxxxx is the least significant byte of the pitch bend value.
Data byte 02	0xxxxxxx, where xxxxxxx is the most significant byte of the pitch bend value. Data bytes 01 and 02 combined can represent 16384 settings.

Channel Mode messages

As you've seen, Channel Mode messages are special cases of Control Change messages. Since they are so important, we'll discuss them individually.

LOCAL ON message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111010=122.
Data byte 02	always 01111111=127.

LOCAL OFF message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111010=122.
Data byte 02	always 00000000=0.

ALL NOTES OFF message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111011=123.
Data byte 02	always 00000000=0.

OMNI OFF message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111100=124.
Data byte 02	always 00000000=0.

OMNI ON message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111101=125.
Data byte 02	always 00000000=0.

MONO ON message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111110=126.
Data byte 02	0xxxxxxx, where xxxxxxx is the number of assigned mono channels from 0 through 15.

POLY ON message	
Status byte	1011xxxx, where xxxx is a channel number from 0 through 15.
Data byte 01	always 01111111=127.
Data byte 02	always 00000000=0.

System Exclusive messages

System Exclusive messages can be freely defined by the manufacturer for a certain hardware device. Only the delimiters are constant.

System Exclusive messages	
Status byte	always 11110000=240.
Data byte 01	0xxxxxxx, where xxxxxxx is the manufacturer identifier.

Data bytes 02-xx	Any number of manufacturer-defined data bytes.
Status byte	always 11110111=247, meaning End of Exclusive message (EOX). This status byte is really a System Common message and has no data byte.

System Common messages

System Common messages control the MIDI system as a whole.

SONG POSITION POINTER message	
Status byte	always 11110010=242.
Data byte 01	0xxxxxxx, where xxxxxxx is the least significant byte of the song position.
Data byte 02	0xxxxxxx, where xxxxxxx is the most significant byte of the song position. Both data bytes 01 and 02 make 16384 addressable song positions.

SONG SELECT message	
Status byte	always 11110011=243.
Data byte 01	0xxxxxxx, where xxxxxxx is a song number from 0 through 127.

TUNE REQUEST message	
Status byte	always 11110110=246.
Data bytes	none

System Real-Time messages
System Real-Time messages consist of only a status byte.

TIMING CLOCK message	
Status byte	always 11111000=248.

START message	
Status byte	always 11111010=250.

CONTINUE message	
Status byte	always 11111011=251.

STOP message	
Status byte	always 11111100=252.

ACTIVE SENSING message	
Status byte	always 11111110=254.

SYSTEM RESET message	
Status byte	always 11111111=255.

This completes the General MIDI 1.0 specifications for MIDI messages.

4.4 Sound Blaster MIDI Hardware

Your Sound Blaster card provides an integrated MIDI interface that conforms to the International MIDI Association (IMA) standard for MIDI devices.

With the optional MIDI connector box or the MIDI adaptor accompanying some versions of Sound Blaster Pro, you can connect any kind of MIDI device to your computer. Therefore, you can use your computer as a sequencer, for example, for composing songs.

You can program your synthesizer on the screen and store programmed songs on your hard disk or floppy disk drive if your keyboard doesn't have a drive of its own.

You'll also want to connect a MIDI keyboard to your computer and transfer songs to your computer system as you play. You can do all

these things with only your Sound Blaster card, the MIDI adaptor or box, a MIDI keyboard, and the appropriate software.

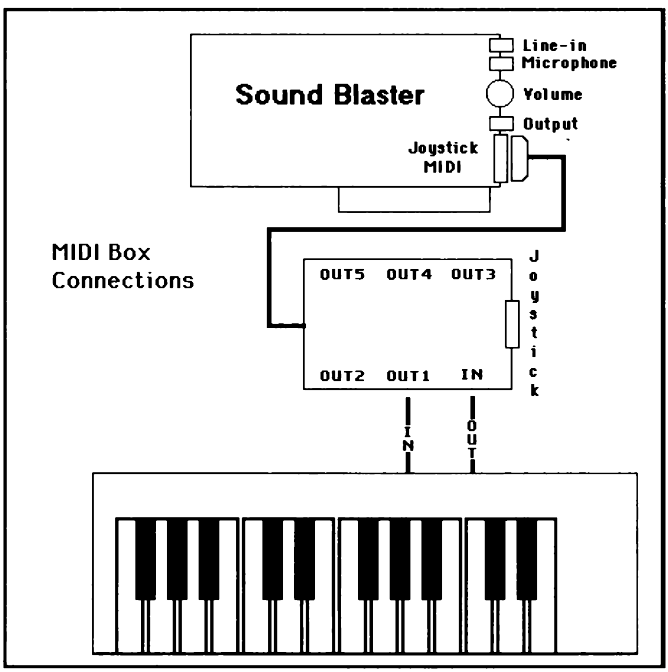
The Sound Blaster MIDI hardware is incompatible with the Roland MPU-401 MIDI interface, so you cannot run MPU-401 software with the Sound Blaster MIDI interface.

Connecting the MIDI hardware

Connecting the MIDI hardware is easy. First, make sure the computer is switched off. If you have a joystick connected to your Sound Blaster card, remove the joystick.

Plug the MIDI hardware into the Sound Blaster card, then plug your joystick into the MIDI hardware's joystick jack. The joystick resumes its normal function there. The MIDI hardware doesn't interfere with the joystick.

If you purchased the MIDI connector box from Creative Labs, you now have five MIDI OUT ports and one MIDI IN port on your computer. Using a suitable cable, hook your MIDI device to the MIDI box (i.e., from MIDI OUT1 of the box to MIDI IN of the device, and from MIDI OUT of the device to MIDI IN of the box).



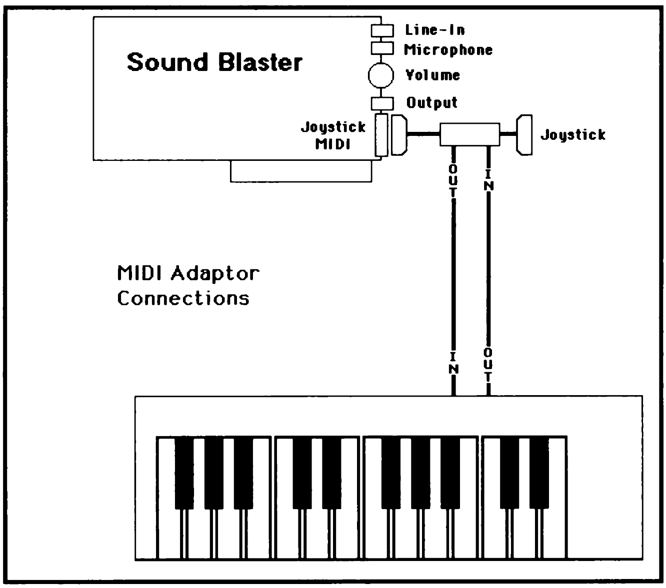
Connecting the MIDI box

Since the MIDI box has only one MIDI IN port, you should connect the keyboard here. The extra MIDI OUT ports can be used for other devices, such as expanders.

Connecting the MIDI adaptor

Connecting the MIDI adaptor is as easy as connecting the MIDI connector box. Again, switch off the computer. Remove the joystick if you have one. Then plug the 15-pin connector of the adaptor into the card.

The MIDI adaptor has one MIDI OUT port and one MIDI IN port. Connect the MIDI OUT plug to MIDI IN of the device, and the MIDI IN plug to MIDI OUT of the device.



Connecting the MIDI adaptor

As with the MIDI connector box, you can now reconnect your joystick using a suitable connector.

After following one of these hardware setups, you're ready to use the MIDI software.

4.5 Voyetra Sequencer MIDI Software

Some Sound Blaster Pro packages contain a complete MIDI Kit, including MIDI sequencer software and the MIDI adaptor. The software is a version of Voyetra Sequencer Junior, which is especially designed for the Sound Blaster Pro. This software is called Voyetra Sequencer Plus.

If you don't have this program, you can use the Intelligent Organ for your first experience with MIDI. It also allows you to record notes from a MIDI keyboard. However, the Intelligent Organ isn't completely suitable for MIDI applications.

Voyetra Sequencer

Voyetra Sequencer, however, is a truly professional program. The Sequencer has 64 tracks and allows you either to send MIDI songs to a connected MIDI device or to use the card itself as a MIDI expander. This means that you can work with the sequencer even if you don't have a MIDI device.

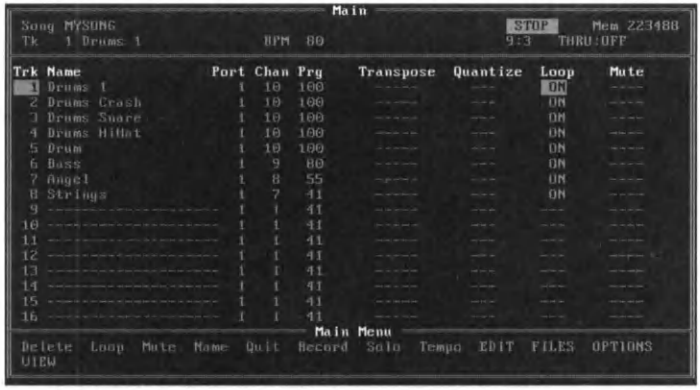
This program may seem unusual and complicated at first, but you'll quickly appreciate its many features. You can use either the keyboard or the mouse to operate Voyetra Sequencer.

The five-part tutorial in the Voyetra manual is a good way to become familiar with the program. It clearly explains all the Sequencer Plus functions.

In this section, we'll introduce the most important functions of this program.

Main screen

The center area of the Main screen shows an excerpt of the 64 tracks with their key parameters. You can scroll this area up or down with either the keyboard or the mouse to check all 64 tracks.



Sequencer's Main screen

Track number The track number is displayed first, on the far left. It's followed by a 20-character name field for describing the track. With the cursor in this field, you can press N and then enter the name.

MIDI or SB ? Next is the port to which the MIDI data of this track should be sent. A 1 indicates that the data should be output over the MIDI interface, while a 2 means that it should be sent to the FM voices of the Sound Blaster card.

MIDI channel Following the port is the MIDI channel number over which the data should be sent. Channel numbers are from 1 to 16.

Program The next heading is the abbreviation "Prg" for program. This refers to the particular preprogrammed sound that must be played. As explained earlier, MIDI program numbering doesn't always agree with the numbering of sounds on your synthesizer.

Transpose The next column specifies whether notes on this track should be transposed. If they should, the transposition is shown in octaves and semi-tones. An up or down arrow indicates the direction of required transposition.

Quantize The column headed "Quantize" identifies a useful Sequencer function. In the quantizing process, notes played slightly out of time can be shifted onto the beat.

You can select the time units to be used. The following settings are available: quarter, eighth, sixteenth, thirty-second, and sixty-fourth notes.

Smaller units give you more subtle control over timing adjustments. Since a certain amount of hesitation or prolongation of notes is part of artistic expression, music that's too strongly quantized can sound sterile.

Loop

The next column indicates whether the track should be played in a loop. If this setting is ON, the track is continuously repeated. Otherwise, it plays once and stops at the end. The L key toggles the Loop status on and off.

Mute, Solo

The last column is labeled "Mute." Sometimes when playing a song, it's useful to silence a particular track while the others continue to play. This can be done by using the M key on the track to be muted. The status MUTE will appear.

You may also want to hear only a single track played as a solo. This is done with the S key on the desired track. All other tracks will immediately be silenced.

Both of these functions can be switched off again the same way they are switched on.

This explains all the columns of track data on the Main screen. Now we'll discuss some of the other functions you can perform from this screen.

Files menu

Let's start by loading a file into memory. You can access the Files menu by pressing the F key.

Within the Files menu, you can use the M key to change modes for file selection. Sequencer Plus can load three types of files. These are song files (ending in .SNG), MIDI files (ending in .MID), and AdLib files (ending in .ROL).

Move the cursor to the desired file, then press L to load the file, and Enter to confirm your choice.

Remember that loading a .SNG file will overwrite one previously loaded in memory. MIDI files, however, can be combined by loading them in succession.

Playing songs

Once you're back in the Main screen, you can start playing the song by pressing the Spacebar.

Some parameters of a song can be changed during play. For example, you can change the sound programs for individual channels and immediately hear the change in instrumentation.

Changing tempo

It's as easy to change a song's tempo. The tempo is shown in beats per minute (BPM) at the top of the screen. The higher the BPM, the faster the song. You can change this value, up to a maximum of 255, after pressing the [T] key.

A general rule for selecting numeric settings is that you can either type the desired number or use special keys to go up or down in increments. The [+] and [-] keys on the numeric keypad will increase and decrease a counter by 1. The [[] and []] keys work in larger steps.

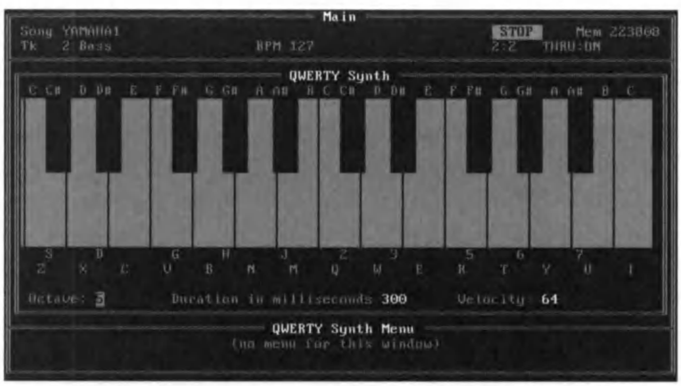
Pressing the [Spacebar] again stops the playing.

Recording

Sequencer Plus also lets you play notes for recording in real-time. This means that you can record onto one track while other tracks are playing. There is a metronome feature in the Options menu for this, which keeps time on the PC speaker and can also give a lead-in count of up to four beats.

You can play the notes in with a MIDI keyboard or with the keyboard of your PC.

To record notes, go to the desired track, set the MIDI channel and program, and press [R]. If you want to use your PC keyboard to input the notes, press [Shift] + [F1] to display the QWERTY Synth keyboard.



Playing notes on the PC keyboard

When you press the [Spacebar], the music will begin and you can start playing. The upper right of the screen shows the remaining available memory.

Recording stops when you press the [Spacebar] again.

View screen

To view recorded tracks, you can move to the View screen by pressing V. On the left, you'll see the key data for each track, while on the right is a graphical representation of the sound data measure by measure. This area can be scrolled to the right to display the full length of the song.

The View screen

From this screen you can add or delete one or more measures, or rearrange them by cutting and pasting.

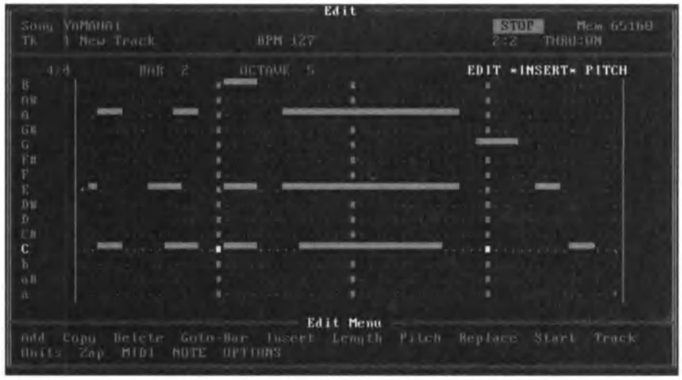
You can also set the starting measure, where playing or recording should begin. From the Main screen, you can start playing and recording only at the beginning of a track.

Edit screen

Once all the desired data is in memory, you can concentrate on the fine points of editing. Don't forget to save your file when something turns out properly. The menu item for saving is also located in the Files menu.

To edit, place the cursor on the track you want to change and press E. The Edit screen appears, showing one measure of the selected track on a modified musical scale.

The pitches that correspond to each line of the scale are displayed vertically along the left edge of the screen. On the scale itself, the notes of the song are represented by bars, which vary in length according to their duration. Although it helps to know how to read standard musical notation, you'll find that the Edit screen provides a good intuitive representation of your music.



A typical Edit screen

You can change the pitch or timing of individual notes by dragging them with the mouse anywhere within the measure. Double-clicking on a note deletes it.

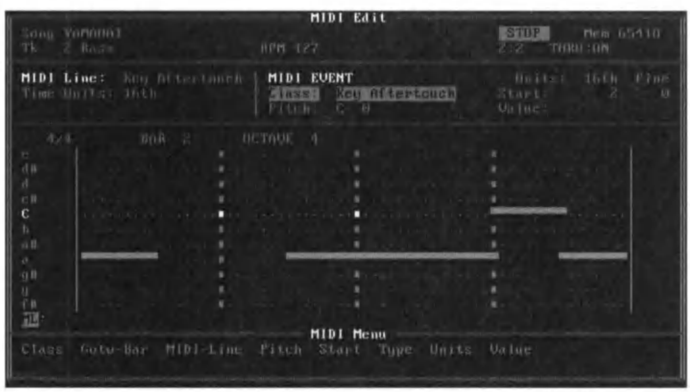
Pressing the Spacebar starts playing at the current measure. This makes it easy to check your work.

Note Edit screen

Other types of note editing can be done from the Note Edit screen, which is accessed by pressing the N key. The top of this screen shows additional information about the note currently selected. All these parameters can also be changed.

MIDI Edit screen

The Esc key returns to the normal Edit mode. From there you can enter the MIDI Edit screen by pressing the M key. This screen allows you to edit MIDI messages.



Inserting MIDI events

Only the Program Change and Pitch Bend messages can be used with your Sound Blaster card acting as an FM expander. For other devices, the other MIDI messages can also be used.

Sequencer Plus is a program that meets the needs of even professional musicians. It has numerous editing capabilities, which are powerful and easy to use.

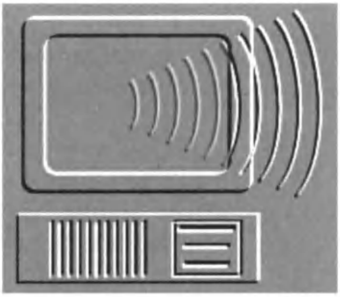
Sequencer Plus Gold

The foundation of the Voyetra Sequencer series, Sequencer Plus Gold, includes some additional features that make it even more useful. One is an editor for the Sound Blaster's FM voices.

It also has 128 FM voices of its own, with which you will certainly be able to create some very fine music.

Sequencer Plus Gold also offers a MIDI file analyzer and a file manager for storing synthesizer data on your hard disk.

Once you're familiar with Sequencer Plus, you'll have no trouble adjusting to Sequencer Plus Gold. The user control interfaces are the same. For a beginner, however, Sequencer Plus is more than adequate and makes learning easy.



Chapter 5
Programming the Sound Blaster Card

Now that you're familiar with the programs for your Sound Blaster card, we'll explain how you can create your own digital sound programs.

All the programs (except one) presented in this chapter are on the companion diskette as source codes and executable files. The VOCSHELL.EXE program is included only as an executable file.

You'll find programs written in Turbo Pascal 6.0, Turbo Pascal for Windows, Borland C++, Borland C++ for Windows, and Microsoft Visual Basic. The detailed documentation of each program will help you easily transfer these examples and function libraries into your favorite programming language.

Before presenting some practical applications, we'll briefly discuss music theory. This will form a basis for your work with digital sound.

5.1 Music Theory

Understanding the mechanics of music

This brief introduction to acoustics and music theory provides some insight into the world of music and its physical characteristics. You'll also learn about a technique called *sampling*. This information is useful for effectively programming your Sound Blaster card.

If you've studied physics or acoustics, this information will probably be familiar to you. However, even if you don't have any experience with physics, you should be able to understand the concepts presented below.

Our discussion of music theory will help you understand what is actually happening when you hear a sound.

What is sound?

Sound

A simple answer to the question "What is sound?" is that a sound is something that you hear. A sound is produced by a source and received by your ear.

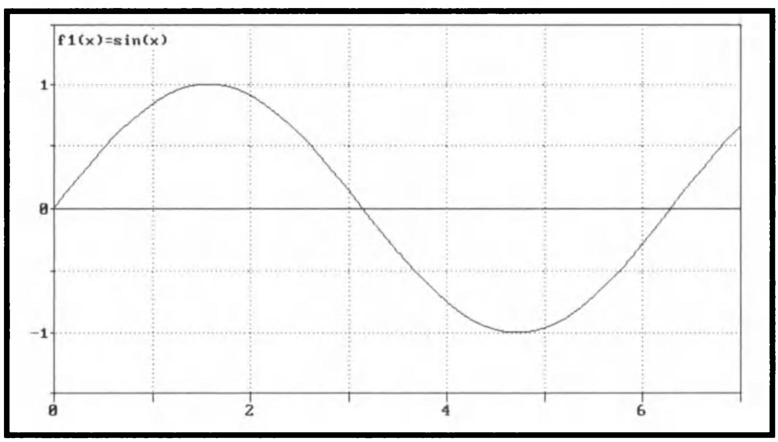
However, for a sound to travel from the sound source to your ear, another element must be available to transmit the sound. This "sound carrier" is called a *medium*. Usually this medium is the air that surrounds us. However, sounds can also travel through water.

Without a medium, sound transmission isn't possible. For example, it's impossible to have a conversation on the moon. Since the moon lacks an atmosphere, a medium isn't present to carry the sounds from your mouth to the listener's ear.

Air is our sound medium

When air pulsates, *sound waves* are created. These waves are then registered as sound. The amplitude and frequency of these waves determine what type of tone is heard.

To clarify this concept, let's look at the most simple type of wave, the sine wave:



A simple sine wave

A sine wave is a periodic wave. This means that its first pulse is followed by an indefinite number of identical pulses. Periodic waves result in sounds that can be called tones, such as the tone of a guitar, a piano, or a bell.

However, the sound of an ocean surf, for example, is considered a noise rather than a tone because it consists of non-periodic waves.

Amplitude of a wave

The primary element of a wave is its strength or *amplitude*. The amplitude is determined by the highest point along the curve of the sound wave. The higher the amplitude, the louder the sound will be. The physical unit of loudness is the decibel (dB). Decibels are a logarithmic unit of measure, specifying the degree of loudness of the wave.

Since this is a complicated concept, we won't discuss it in detail. However, you don't have to know a lot about decibels to program your Sound Blaster card. Simply remember that the loudness of a sound is changed by varying the amplitude of its wave.

Frequency of a wave

The second element of a wave is its frequency. How high or low a given tone sounds depends on the number of pulses per second. This number of pulses is referred to as the tone's *frequency*. The unit of measure for frequency is Hertz (Hz). This unit specifies the number of pulses per second that a given tone emits.

The higher a tone's frequency, the higher the tone will sound. A tone that emits 440 pulses each second has a frequency of 440 Hz.

Concert A

In the music world, the frequency of 440 Hz is very important. This frequency defines the concert pitch a1, according to which most concert instruments are tuned. This frequency will appear in several other examples.

The human ear is capable of perceiving sounds between 20 Hz and 20000 Hz. However, this range varies considerably from person to person, particularly at the high-frequency limit. The ability to hear these upper frequencies decreases with age.

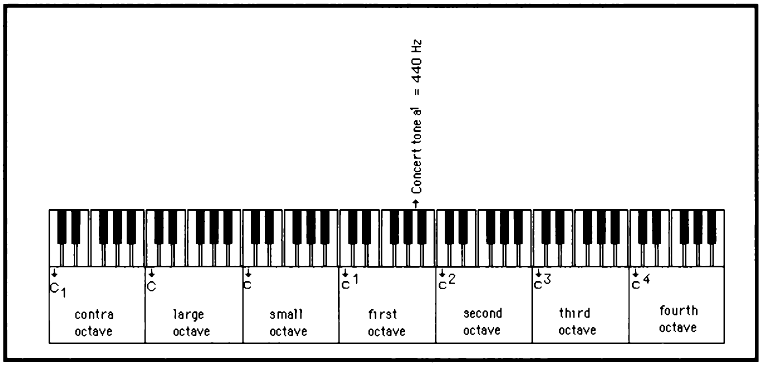
The most useful frequency range lies below 10000 Hz. Almost all frequencies connected with music, speech, and noises are located within this range. To give you a general idea of what different frequency ranges actually mean, we've included the following table, which lists the ranges of various sound sources:

Instrument	Frequency range
Human voice	70 - 2000 Hz
Pipe organ	16 - 4000 Hz
Piano	30 - 3500 Hz
Violin	200 - 3000 Hz
Flute	260 - 3000 Hz

Now that we've discussed the two factors that determine the characteristics of a tone, we'll discuss music theory.

From a frequency to a tone...

A tone with a frequency of 220 Hz and another with a frequency of 880 Hz are referred to as "A". However, the former lies an octave below concert A, while the latter lies an octave above concert A.



A keyboard with the tone a1

This arrangement of frequencies illustrates a basic musical principle: A doubled frequency results in exactly the same note, only one octave higher.

Therefore, the frequency ratio between the base tones of two successive octaves is 1:2. The relationship between the notes or tones within an octave can be described in the same way.

Intervals

The following are the frequency ratios between the various intervals:

Intervals	Notes	Example
Root	1 : 1	c - c
Minor third	5 : 6	c - e flat
Major third	4 : 5	c - e
Fourth	3 : 4	c - f
Fifth	2 : 3	c - g
Minor sixth	5 : 8	c - a flat
Major sixth	3 : 5	c - a
Octave	1 : 2	c - c'

Musical scales

The modern musical scale consists of seven base tones and five half tones. The seven base tones are located either a whole step from the adjacent note (C-D, D-E, F-G, G-A, A-B) or a half step from the adjacent note (E-F, B-C). Each note can be raised or lowered by a half step.

When written on sheet music, a raised note is preceded by a sharp (#), and a lowered note is preceded by a flat (b).

However, this results in a minor problem. If we take a mathematical approach, we discover that a D sharp (D#) and E flat (Eb) don't possess exactly the same frequency. So, theoretically, a keyboard must contain two separate keys for these notes.

Instruments equipped with these extra keys actually have been built, such as a harpsichord created by Marin Mersenne (1588-1648).

Scale models

At that time two old scale models were still being used. One was based on the Pythagorean principle and the other on the diatonic principle. Although both of these models had been worked out very precisely mathematically, often they weren't suitable for musical purposes.

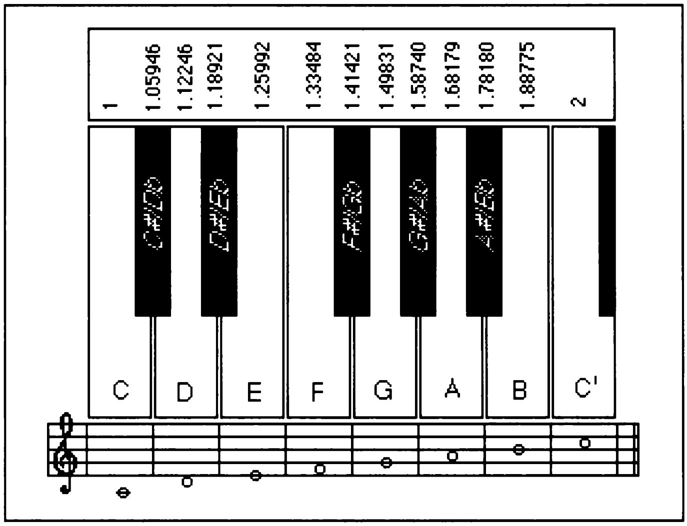
As a result, the *tempered scale* was developed. In this scale, the intervals between the half steps are always equal. This means that D sharp and E flat actually have the same frequency and can therefore be played on the same key.

So today keyboard instruments have only 12 notes per octave. The discrepancies in frequency that occur because of this compromise are so minimal that they aren't noticeable. Also, musicians playing string instruments, such as a violin, have the option of still playing D sharp differently than E flat, since the notes of these instruments aren't bound to specific keys or frets.

Since this option still exists, the system of raising and lowering notes through sharps and flats has been retained. Otherwise, one of the two notations could have been eliminated, while still achieving the same musical results.

The following illustration represents a normal keyboard with a 12 note scale. The frequency ratios between the individual notes are shown above the keys. This means that to obtain a g sharp, for

example, you must perform the following calculation: frequency (G sharp) = frequency (C) * 1.58740.



Frequency ratios of the tempered 12 tone scale

These frequency ratios correspond to those of the tempered scale. Each note is exactly one "12th root of 2" higher than its predecessor (˜ 1.0595).

This completes our discussion of musical scales. Now you should have enough knowledge to be able to write your own music program using the principle of the 12 tone scale.

The diversity of sound

Besides defining sound, we must also understand why there are so many different kinds of sounds and how these sounds are produced.

Each instrument has a different characteristic sound. For example, if you play a concert A on a guitar, its frequency is 440 Hz. If you play the same note on a flute, its frequency is also 440 Hz but it sounds entirely different.

This occurs because the timbre (pronounced "TAM ber") or sound of an instrument consists of multiple waves, unlike the sine wave example shown above.

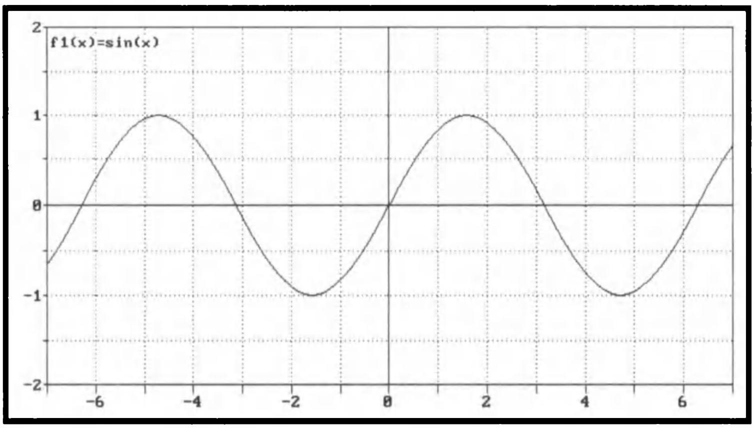
Overtones determine the sound

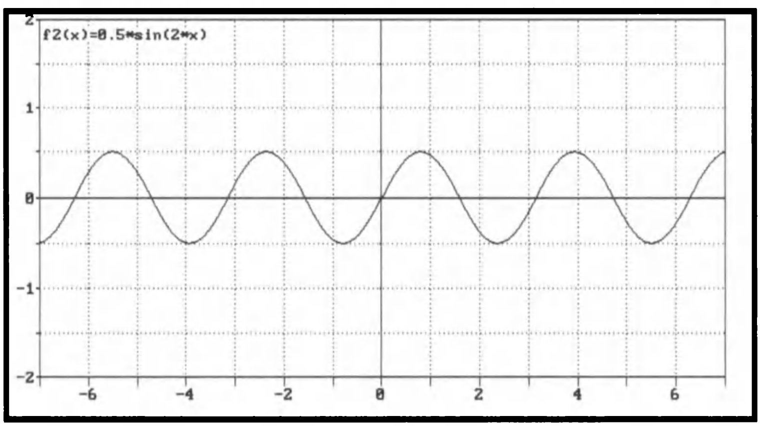
Each instrument emits a specific number of overtones, which determine the characteristics of its sound. Overtones are sound waves, or frequencies, that occur as multiples of the tone's basic frequency.

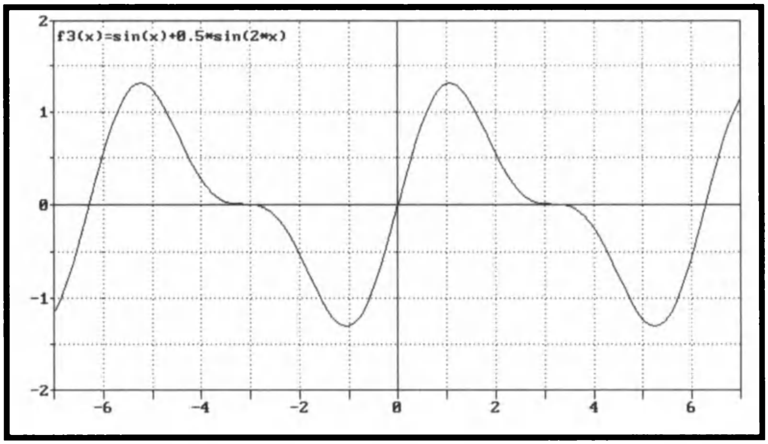
As an example, again let's use an instrument that's playing a tone with a frequency of 440 Hz. At the same time, the instrument could be emitting 880 Hz, 1320 Hz, 1760 Hz, and also another half dozen multiples of 440 Hz.

These overtones produce the sounds of different instruments and their individual characteristics. However, the basic frequency always determines the pitch of the tone.

As with the basic frequency, the amplitudes of the individual overtones also determine how the tone sounds. So, theoretically an infinite number of variations on the tone with the basic frequency of 440 Hz are possible.







Superimposing a basic tone with an overtone

Now we must determine how to produce natural sounds with artificial sound sources.

This is especially important if you want to create sounds using the FM voices of your Sound Blaster card or if you want to manipulate sound samples mathematically.

Fourier synthesis

At the beginning of the eighteenth century, the French mathematician Jean Baptiste Fourier had explored how a natural sound could be broken down into basically identical elements. For this he developed a mathematical process that divides a sound into a finite number of different sine waves. This process is known as Fourier analysis.

This process also works in reverse, so Fourier synthesis can be used to produce specific sounds artificially. For example, it's possible to produce an instrument sound by taking a sine wave as a base tone and adding the required number of overtones. In this way you can digitally produce a natural sound without having previously recorded the sound (i.e., sound sampling).

The biggest disadvantage of this process is that you must calculate and add a separate wave form for each individual pulse of a sound. So a tremendous amount of calculation is needed to produce a "real" sound.

Although the number of different frequencies and their amplitude sequences result in a large amount of data that must be processed, this process is still frequently used.

Instead of creating natural sounds, you can also record sounds and then manipulate them digitally.

Sampling

The process of digitizing sounds is called *sampling*. This method allows a sound, which consists of an analog signal, to be transformed into digital data (i.e., bits and bytes). This task is performed by an Analog to Digital Converter (ADC).

Analog to Digital Converter

The ADC scans or samples the analog signal at predetermined intervals and converts the results of these samples into numeric values.

The interval at which this occurs is determined by the sampling rate, which is also called the sampling frequency. The unit of measure for this frequency or rate is Hertz, just like the unit for measuring the frequency of a tone.

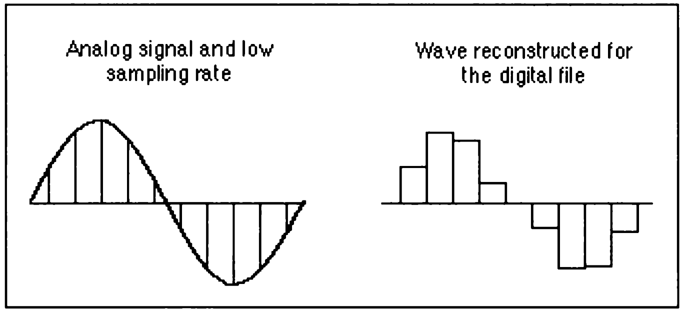
At a sampling rate of 4000 Hz, the ADC samples the analog signal 4000 times each second. This means that 4000 8-bit values, which is equal to 4000 bytes, are stored each second.

So, with a sampling rate of 44100 Hz, 44100 bytes containing information about the analog signal are stored each second.

The enormous mass of data that's accumulated in this process clearly shows why sample files usually occupy a lot of disk space. Therefore, you must compromise between a manageable amount of data and a sufficient sample quality.

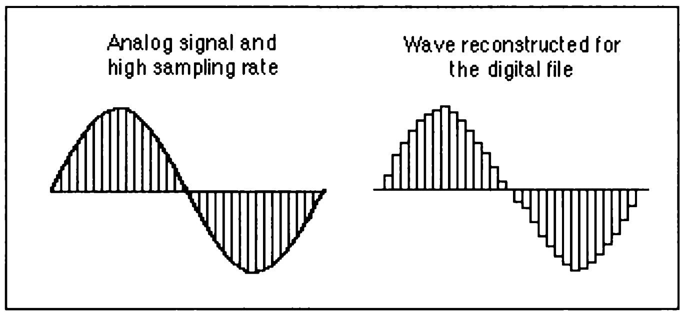
This is necessary because the sample quality increases with the sampling rate, resulting in a digital signal that's truer to the analog original.

The following illustration show an analog signal sampled at a low sampling rate:



An analog signal sampled at a low sampling rate

The following illustration show an analog signal sampled at a high sampling rate:



An analog signal sampled at a high sampling rate

An audio CD, for example, uses a sampling rate of 44100 Hz per second, using 16-bit values. So this process requires 88200 bytes of data per second for each stereo channel.

This means that for each second of stereo CD sound you hear, 176400 bytes are processed. If you consider that a single audio CD can contain slightly more than 70 minutes of stereo music, you'll quickly realize how much information can be stored on these little disks.

When choosing the best sampling rate, the compromise also depends on the particular sound that will be sampled. For example, suppose that you're digitizing the sound of the ocean surf. By using a lower sample frequency, a hiss is produced. However, this hiss is barely noticeable. Usually you can find the best solution by experimenting.

Shannon theorem

You should follow a formula created by the mathematician Shannon when you're selecting a sampling rate. According to Shannon, the sample frequency must be exactly double the highest sound frequency that will be digitized. If this requirement is fulfilled, all necessary information on the analog signal will be stored digitally.

This means that if you want to sample a frequency of 12000 Hz, you must use a sampling rate of 24000 Hz to obtain optimum results.

As we mentioned, the upper limit of human hearing lies around 20000 Hz. So an audio CD, with its 44100 Hz sampling rate, achieves near perfect quality. This sampling rate can be used to digitize the entire audio spectrum.

Digital to Analog Converter

When the sample is played back, a Digital to Analog Converter (DAC) reverses the conversion that was previously made by the ADC. The DAC converts the digital data back into an analog signal. However, instead of being smooth and continuous like the original analog sound, this reproduced analog signal consists of individual steps. This can be seen in the previous illustration, which shows how the ADC works.

When a digitized sound is played back with the sampling rate with which it was recorded, it sounds normal. If the sampling rate is lowered, the reproduced sound is slower and lower.

If the sample is played back at a higher sampling rate, the reproduced sound is faster and, therefore, higher. You can use this trick to manipulate a human digitized voice.

You can also use this technique to achieve some interesting effects with sampled instrument sounds. For example, if you've sampled concert A (a1) with a sampling rate of 10 KHz and then play back the sample using a sampling rate of 20 KHz, the tone will be exactly one octave above the original A, as we mentioned.

However, it isn't feasible to increase the pitch of a sequence of tones by an octave in this way. If you do this, not only the frequency, but also the speed of the samples are changed. So the digitized sound will sound only half as long because its data is being played at double the original rate. However, you may find other uses for this technique.

5.2 Digital Sound Channel

In this section we'll show you how you can program your Sound Blaster card's digital sound channel.

For DOS, you'll find programs written in Turbo Pascal 6.0 and Borland C++; drivers supplied by Creative Labs are used for the sample output.

For Windows, you'll find programs written in Turbo Pascal for Windows, Borland C++ for Windows, and Visual Basic. The sample format used in these examples is the standard Windows WAV sample format.

We'll also explain how the various driver programs are addressed, so you'll be able to use these drivers with other programming languages.

In the following sections we'll describe the various characteristics of your Sound Blaster driver and the sample format. With this information, you'll be able to program your sound card from DOS.

5.2.1 Structure of CT-Voice format

CT-Voice format

In addition to the Sound Blaster card, Creative Labs has also introduced a new format for digitized sounds. This format is called the Creative Voice File format, which is indicated by the .VOC extension.

So you can completely understand the individual functions of the Creative Labs driver software, first we'll describe this special format.

A VOC file is divided into two blocks: the header and the actual data.

Header

CT-Voice header block

The header is a data block that identifies the file as a CT-format file. This means that you can use the header to check whether the file is an actual CT-format file.

Bytes $00 - $13 (0 - 19)

The first 19 bytes of a VOC file contain the text "Creative Voice File", as well as a byte with the value $1A. This identifies the file as a VOC file.

Bytes $14 - $15 (20 - 21)

These bytes contain the offset address of the sample data as a low-byte/high-byte value. At this point, this value is $001A because the header is exactly $1A bytes long.

However, if the length of the header changes later, the programs that access the VOC data in this file will be able to use the values stored in these two bytes to determine the location at which the sample data begins.

Bytes $16 - $17 (22 - 23)

These two bytes contain the CT-Voice format version number as a low-byte/high-byte value.

The current version number is still 1.10 so byte $17 contains the main version number ($01) and byte $16 contains the version sub-number ($0A). The version number is very important because later CT-Voice format versions may use an entirely different method for storing the sample data than the current version.

To ensure that the data contained in the file will be processed correctly, you should always check the file's version number. If a different version number appears, an appropriate warning is displayed.

Bytes $18 - $19 (24 - 25)

The importance of the version number is obvious in bytes $18 and $19. These bytes contain the complement of the version number, added to $1234, as a low-byte/high-byte value.

Therefore, with the current version number $010A, byte $18 contains the value $29, while byte $19 contains $11. This results in the word value $1129. If you check this value and successfully compare it to the version number stored in the previous two bytes, you can be almost certain that you're using a VOC file.

This completes the description of the bytes contained in the header. Everything that follows these bytes in the file belongs to the file's data blocks.

The data blocks

The eight data blocks of the CT-Voice format have the same structure, except for block 0.

Each block begins with a block identifier, which is a byte containing a block-type number between 0 and 7. This number is followed by three bytes specifying the length of the block, and then the specified number of additional data.

The three length bytes contain increasing values (i.e., the first byte represents the lowest value and the third byte represents the highest). So the block's length can be calculated by using the formula:

```
Byte1 + Byte2*256 + Byte3*65536
```

In all other cases, the CT-Voice format stores values requiring more than one byte in a low-byte followed by a high-byte, which corresponds to the word data type.

Block 0 - End Block

The end block has the lowest block number. It indicates there aren't any additional data blocks. When such a block is reached, the output of VOC data during the playback of digitized sounds stops. Therefore, this block should be located only at the end of a VOC file. The end block is the only block that doesn't have bytes indicating its block length.

Structure of the End Block	
Block Type	1 byte = 0
Block Length	none
Data Bytes	none

Block 1 - New Voice Block

The block type number 1 is the most frequently used block type. It contains playable sample data. The three block length bytes are followed by a byte specifying the sampling rate (SR) that was used to record the sounds.

Calculating the sampling rate

Since only 256 different values can be stored in a single byte, the actual sampling rate must be calculated from the value of this byte.

Use the following formula to do this:

```
Actual_sampling_rate = -1000000 DIV (SR - 256)
```

To convert a sampling rate into the corresponding byte value, reverse the equation:

```
SR = 256 - 1000000 DIV actual_sampling_rate
```

The pack byte follows the SR byte. This value indicates whether and how the sample data have been packed.

The value 0 indicates that the data haven't been packed; so 8 bits form one data value. This is the standard recording format. However, your Sound Blaster card is also capable of packing data on a hardware level.

A value of 1 in the pack byte indicates that the original 8 bit values have been packed to 4 bits. This results in a pack rate of 2:1. Although the data require only half as much memory, this method also sacrifices some sound quality.

The value 2 indicates a pack rate of 3:1, so the data require only a third of the memory. Unfortunately, the sound quality decreases significantly.

A pack byte value of 3 indicates a pack rate of 4:1, so 8 original bits have been packed down to 2. This pack rate results in a considerable loss of sound quality.

The pack byte is followed by the actual sample data. The values contained in the block length bytes also indicate the length of the sample data. To determine the length of the actual sample data in bytes, simply subtract the SR and pack bytes from the block length.

Structure of the New Voice Block	
Block Type	1 byte = 1
Block Length	3 bytes
SR Byte	1 byte
Pack Byte	1 byte = 0,1,2,3
Data Bytes	x bytes

Block 2 - Subsequent Voice Block

Block type 2 is used to divide sample data into smaller individual blocks. This method is used by the Creative Labs Voice Editor when you want to work with a sample block that's too large to fit

into memory in one piece. This block is then simply divided into several smaller blocks.

Since these blocks contain only three length bytes and the actual sample data, blocks of type 2 must always be preceded by a block of type 1. So, the sampling rate and the pack rate are determined by the preceding block type 1.

Structure of the Subsequent Voice Block	
Block Type	1 byte = 2
Block Length	3 bytes
Data Bytes	x bytes

Block 3 - Silence Block

Block type 3 uses a small number of bytes to represent a mass of zeros. First there are the three familiar block length bytes. The length of a silence block is always 3, so the lowest byte contains a three, and the other two bytes contain zeros.

The length bytes are followed by two other bytes, which indicate how many zero bytes should be replaced by the silence block.

This is followed by a byte that indicates the sampling rate for the silence block. The SR byte is encoded in the same way as indicated in block type 1.

Silence blocks can be used to insert longer pauses or silences in a sample, which reduces the required data to a few bytes. The Voice Editor will insert these silence blocks through the Silence Packing function.

Structure of the Silence Block	
Block Type	1 byte = 3
Block Length	3 bytes = 3
Duration	2 bytes
Sample Rate	1 byte

Block 4 - Marker Block

The marker block is an important element of the CT-Voice format. It also has three block length bytes followed by two marker bytes.

The block length bytes always contain the value 2 in the lowest byte.

When the playback routine of "CT-VOICE.DRV" encounters a marker block, the value of the marker byte is copied to a memory location that was specified to the driver.

In our programs in Turbo Pascal, this will be the variable "VoiceStatusWord". You can use this function within your programs to determine the place at which your sound playback is located in the sample. You can then, for example, adjust the action of your program to correspond with the sound playback. This function is very useful when you're synchronizing graphics and the playback of a sample.

Using the Voice Editor, you can divide large sample data blocks into smaller ones, inserting marker blocks at important locations. This doesn't affect the playback of the sample. However, you'll be able to determine, from your program, which point of the sample the playback routine is currently reading.

Structure of the Marker Block	
Block Type	1 byte = 4
Block Length	3 bytes = 2
Marker	2 bytes

Block 5 - Message Block

It's also possible to insert ASCII texts within a VOC file. Use the message block to do this. If you want to identify a specific section of a sample file by adding a title, simply add a block of type 5, in which you can then store the desired text.

This block also has three block length bytes. These bytes are followed by the text in ASCII format. The text must contain a 0 in the last byte to indicate the end of the text. This corresponds to the string convention of the C programming language. This allows you to print the texts in a VOC file directly from memory using the `printf()` function.

Structure of the Message Block	
Block Type	1 byte = 5
Block Length	3 bytes
ASCII Data	x bytes
End Character	1 byte = 0

Block 6 - Repeat Block

Another special characteristic of the CT-format is that it's possible to specify, within a VOC file, whether specific sample sequences should be repeated. Blocks 6 and 7 are used to do this.

Block 6 has three block length bytes, followed by two bytes indicating how often the following data block should be repeated. If the value specified here is 4, the next block is played a total of five times (one "normal" playback and four repeats).

Structure of the Repeat Block	
Block Type	1 byte = 6
Block Length	3 bytes = 2
Counter	2 bytes

Block 7 - Repeat End Block

Block 7 indicates that all blocks between block 6 and block 7 should be repeated. With this block, several data blocks can be included in a repeat loop. However, nested loops aren't allowed. The driver is capable of handling only one loop level.

Block type 7 also has three block length bytes, which actually aren't necessary because this block doesn't contain any additional data. Therefore, the block length is always 0.

Structure of the Repeat End Block	
Block Type	1 byte = 7
Block Length	3 bytes = 0

Block 8 - Extended Header Block

This block had to be introduced with SB Pro because the normal data block 1 couldn't manage information that could be used to make statements about stereo data.

Block type 8 solves this problem. Whenever this block precedes a block type 1, all the information about the sample data from block type 8 is taken instead of the actual information from block type 1. However, the sample data still comes from block type 1.

It may have been more logical to create a real block type 8 instead of using this "8+1" solution. This real block 8 could also contain the sample data, but this procedure would allow older driver versions, which don't recognize block type 8, to output data. However, interpreting the data from defective information would probably produce strange results.

Block type 8's structure is very similar to the information section of block type 1. The converted sample rate follows the three length bytes, which contain a constant value of 4. In this case, the sample rate is converted to 16 bit, or 2 bytes.

The sampling rate (SR) is also calculated differently than with block type 1. The formula is now SR = 65536 - (256000000 DIV effective sample rate). For stereo samples, the effective sample rate must be doubled. Therefore, the formula for stereo samples is SR = 65536 - (256000000 DIV (effective sample rate *2)).

The pack byte is the next byte in block type 8. The interpretation is identical to block type 1. Simply remember that a stereo sample must be unpacked.

New is the next byte. This byte describes the mode of the sample data. A 0 in this byte indicates a mono sample; a 1 indicates that the data should be interpreted as a stereo sample.

Structure of the Extended Header Block	
Block Type	1 byte = 8
Block Length	3 bytes = 4
Sample Rate	2 bytes
Data Mode	1 byte

We've now described the different block types used in VOC files. These functions are fully supported by the CT-VOICE.DRV driver software.

If you'll be writing your own sound programs, you should follow this format because it's easy to use and flexible. When needed, new block types are easily added. Programs that don't recognize block types should be written so they continue operating after an unrecognized block. This is easy to do because each function specifies its own block length.

The CT-VOICE.DRV driver is a small but extremely powerful software tool that allows you to access all the important functions of your sound channel.

You can easily link this driver to your own programs, whether they are in assembly or another high-level language, such as Turbo Pascal.

All driver functions are tailored specifically to the CT-Voice format of Creative Labs.

When loading the CT-VOICE.DRV file, remember that the driver can be loaded to any particular segment start. This means that the offset address of the driver start must be $0000 because all jumps within the driver depend on this address.

CT driver functions

Driver functions are then called by jumping to this starting address of the driver. All parameters that should be passed to the driver are exchanged through the processor register. Function results are returned in the same way.

BX is the most important register for a function call. The number of the desired function is specified in this register. Also, registers AX and DX are modified from within the driver routines.

For example, register AX might be used to return values to the program that triggered the function call. All registers, except for AX and DX, remain unchanged through function calls. This also applies to the flags register.

Function 0 (BX=0): Determine driver version

This function returns the version number of the CT-VOICE.DRV you're currently using. After the function call, this number is stored in AX register, whereby AH represents the main number and AL

represents the subversion number. The current version number, for example, is 1.10.

This means that register AX contains the value $010A, whereby AH contains the main number ($01 = 1 decimal) and AL the subversion number ($0A = 10 decimal).

As new driver versions, which provide more functions than the 13 functions included in the current version, are introduced, the version number must be verified. If your program tries to use a new function, first you must verify that the driver that was loaded is the proper version.

This is particularly important because the SB Pro driver won't function properly with Sound Blaster Versions 1.0 through 2.0. However, the drivers used for Versions 1.0 through 2.0 are upwardly compatible with SB Pro.

Determine driver version	
Input	BX = 00
Output	AH = Main number
	AL = Sub-number
Remarks	none

Function 1 (BX=1): Set port address

The standard port address of the driver is indicated in the CT-VOICE.DRV file. It's located at offset $30 and is of the type word (2 bytes).

If function 1 isn't called before the driver is initialized, the value found at this location is used.

However, if the driver must be started with another port address, you must specify a new base address in register AX with the function 1 call. This must be done before the card is initialized with function 3; otherwise, the function remains ineffective.

The correct value for your card depends on the jumper settings of your Sound Blaster card. The values $210, $220, $230, $240, $250, and $260 or $220 and $240 are acceptable, depending on the version of your card. For more information, refer to the sections, in Chapter 1, on the settings and the installation of the card.

You can use the INST-DRV.EXE program to determine the correct address in your CT-VOICE.DRV file. This program automatically writes the address permanently into the file. So, as we mentioned in Chapter 1, if you've made any changes to the manufacturer's default settings, this modification should be made after the card has been installed.

Set port address	
Input	BX = 01
	AX = Port address
Output	none
Remarks	can only be called before function 3

Function 2 (BX=2): Set interrupt

This function is similar to function 1. You'll find the default interrupt number in CT-VOICE.DRV, at offset $32, in the form of a byte. If function 2 isn't called before the driver is initialized, the value found here is used.

If you want to change the interrupt number, you must specify the desired value in register AX with the function call. The values 2,3,5, and 7 or 2,5,7, and 10 are allowed, depending on the version of your card and its jumper settings. For more information refer to Chapter 1.

Again, you can make a permanent change in your CT-VOICE.DRV file by using INST-DRV.EXE.

Set interrupt	
Input	BX = 02
	AX = Interrupt number
Output	none
Remarks	can only be called before function 3

Function 3 (BX=3): Initialize driver

This function initializes your sound card for all subsequent functions. If you will be using functions 1 and 2, they must be called before this function is executed.

The initialization procedure checks whether all parameters have been set correctly and returns a corresponding value in the AX register.

If AX=0, an error hasn't occurred and the port address as well as the IRQ number are correct.

If AX=1, the procedure was unable to find the Sound Blaster card.

AX=2 indicates an I/O error, which means that an incorrect port address has been set.

AX=3 indicates an interrupt error, which means that an incorrect interrupt number has been set.

After the driver has been initialized successfully, the speaker output is automatically switched on so a sample output can be created.

All subsequent function numbers may be called only after function 3 has been executed successfully.

Initialize driver	
Input	BX = 03
Output	AX = 0 Successful
	AX = 1 SB not found
	AX = 2 Port address error
	AX = 3 Interrupt error
Remarks	none

Function 4 (BX=4): Loudspeaker on/off

Function 4 either switches the loudspeaker output of your Sound Blaster card on or off, depending on the value passed in the AX register.

If AX=0, the speaker is switched off. For all values not equal to 0, the speaker is switched on.

Function 4 is called automatically by function 3 (Initialize) to switch on the system and by function 9 (De-install) to cut the speaker connection.

Loudspeaker on/off	
Input	BX = 04
	AL = 0 off
	AL = 1 on
Output	none
Remarks	none

Function 5 (BX=5): Set StatusWord address

This function allows the program that's using the driver to control the actions of your Sound Blaster card.

The Offset:Address of a WORD type variable is passed to the driver in the register pair ES:DI. The status of the current sample will always be written to this location. A value other than 0 indicates that the sample is currently being processed (either recorded or played).

If the value of StatusWord is 0, a sample isn't currently being processed or the sample has already been processed.

The availability of such a status variable is particularly important because functions, such as recording or playing samples, immediately return control to the program that called them when they start.

During playback of samples, the values of marker blocks encountered within the samples are copied to the status variable.

Set StatusWord address	
Input	BX = 05
	ES:DI = Status address
Output	none
Remarks	none

Function 6 (BX=6): Sample playback

This function starts the output of a recorded sample, which must be located in memory and must conform to the Creative Voice File Format. You'll find a detailed description of this format in the section on the CT-Voice format.

The register pair ES:DI passes the Segment:Offset of the sample's first data block. This means that the file header must be jumped over.

At the start of the output routine, StatusWord is immediately set to $FFFF, function 6 is ended, and output or playback is continued independently.

This is possible because the Sound Blaster card can directly access the PC's memory. This access takes place through the DMA channel. So, as long as you don't perform any actions that also use the DMA channel, your program can continue without any interruptions.

Don't try to start the playback or recording of a sample if another sample is already being played back. First you must interrupt or terminate the first playback. If you don't do this, a system crash will probably occur.

Sample playback	
Input	BX = 06
	ES:DI = Sample address
Output	none
Remarks	StatusWord is changed according to encountered marker block

Function 7 (BX=7): Record sample

This function allows you to transfer digital sounds into memory through your DMA channel.

Function 7 requires several parameters. The AX register must contain the sampling rate, whereby values between 4,000 Hz and 44,100 Hz are permitted. This range also depends on the version of your sound card.

Also, the sample's maximum length must be specified (variable type may be either LONG or DOUBLE WORD). Use the register pair DX:CX to do this.

Similar to function 6 (Playback), the address of the sample memory must also be specified here. This is necessary so the driver will know to which memory locations it must write the transmitted values.

This address is passed through the register pair ES:DI (Offset:Address).

Once the function has been called, control is again returned to the program that performed the function call. Here the same conditions apply as in the case of function 6.

StatusWord contains a value other than 0 as long as the recording is in progress. Once the input buffer is full, or rather once the data length specified in DX:CX has been reached, it is set back to 0.

Record sample	
Input	BX = 07 AX = Sampling rate DX:CX = Length ES:DI = Sample address
Output	none
Remarks	none

Function 8 (BX=8): Abort sample

This function interrupts all running sound card activities, such as the recording or playback of samples. StatusWord is reset to 0. Additional parameters aren't required.

Abort sample	
Input	BX = 08
Output	none
Remarks	StatusWord = 0

Function 9 (BX=9): De-install driver

This function de-installs the driver software so the sound card is effectively returned to its power-up condition.

The speaker connection is automatically switched back off. Additional parameters aren't required.

De-install driver	
Input	BX = 09
Output	none
Remarks	none

Function 10 (BX=10): Pause sample

This function pauses the output of a sample that's currently being played back. StatusWord remains unequal to 0 because the playback isn't completed yet.

In the AX register, the value 0 is returned if the pause was successful. However, if AX contains the value 1, a sample wasn't being played back when the function was called.

Pause sample	
Input	BX = 10
Output	AX = 0 Successful
	AX = 1 Not successful
Remarks	StatusWord remains unchanged

Function 11 (BX=11): Continue sample

This function is the counter function to number 10. You can use it to continue a paused sample playback.

If the value 0 is returned in register AX, the playback was continued successfully. If it contains 1, a sample wasn't paused.

Continue sample	
Input	BX = 11
Output	AX = 0 Successful
	AX = 1 Not successful
Remarks	none

Function 12 (BX=12): Interrupt loop

This function allows you to interrupt the loop functions supported by the Creative Voice File format. You'll find a more detailed description of these loop functions under the description of the Creative Voice File format.

Depending on the value specified in the AX register, there are two ways to interrupt a loop. If AX = 0, the current pass of the loop will be completed, after which playback will continue with the next data block following the loop.

If AX = 1, the loop will be interrupted immediately and playback will continue with the next data block following the loop end block.

This function will return the value 0 in register AX if the loop was interrupted successfully. The value 1 is returned if a loop wasn't being executed at the time of the function call.

Interrupt loop	
Input	BX = 12
	AX = 0 at end of loop
	AX = 1 immediately
Output	AX = 0 Successful
	AX = 1 no loop being executed
Remarks	none

Function 13 (BX=13): User-defined driver function

This function allows you to add a new function to your driver, which is called automatically when a new data block is reached in a sample that's being played back.

Then this function or procedure can be integrated into the automatic process of the playback of sound samples.

However, to design this type of user function, several specific requirements must be met. So we recommend doing this only if you're an experienced programmer.

Function requirements

User-defined functions must be designed according to the following requirements:

- The contents of all processor registers must be saved and, upon completion of the function, be restored to their respective registers. The only exception to this rule is the carry flag in the FLAGS register. This flag can be used by your function to send a command to the driver.

  Carry = 0 indicates that the current data block must be played. Carry = 1 specifies that the block shouldn't be played.

  Remember, the carry flag should never be set for block type 0 (sample end block) because the driver automatically terminates playback upon reaching block type 0.

- The function must be ended with a FAR RET instruction.

  To activate the user function, its Segment:Offset address is passed in the register pair DX:AX at the call of function 13. If the function must be deleted, DX and AX must be set to 0.

  At the time the user function is called by the driver, the register pair ES:BX will point to the first byte of the current data block. This is the location of the block type definition.

User-defined function	
Input	BX = 13
	DX:AX Function address
Output	ES:BX Address of the current data block
Remarks	none

This completes our discussion of the driver functions. These functions provide everything you need to successfully program the digital channel of your Sound Blaster card.

The descriptions we've presented provide a quick overview of these functions.

5.2.3 VOC Programming with Turbo Pascal 6.0

If you use assembly language, the description of the driver's memory requirements and its individual functions and registers should help you work with this program.

Now we'll discuss how this driver is used from within a high-level programming language. We'll use Version 6.0 of Turbo

Pascal to demonstrate how you can implement the driver effectively.

The VOCTOOL.PAS unit forms the core of the program that we'll present. It will act as the interface between the driver and the programming language.

The VOCTOOL.TPU unit uses an interesting property of the Turbo Pascal unit-system, called the unit initialization section.

In most units, this section simply consists of a Begin/End construct. However, in the VOCTOOL.TPU unit much is accomplished between these statements.

Automatic initialization

First the unit's ExitProc is transferred to a new procedure, called VoiceToolsExitProc. This ensures that the program ends properly.

The ExitProc is the procedure that's automatically executed at the end of the program, as the initialization section of each linked unit is executed at the beginning.

VOCFile-Header

Then the most important variables are initialized. VOCFileHeader is a new variable provided by the VOCTOOL.TPU unit. It contains the byte sequence stored at the beginning of each VOC file.

This variable isn't used within the unit. However, it may be extremely useful for your own programs (e.g., when you want to create your own VOC file that must be saved in the CT-Voice format).

VOCDriver-Installed

Next the VOCInitDriver function is called. If its result isn't TRUE, the variable VOCDriverInstalled receives the value FALSE because an error has occurred during the initialization attempt. However, if the function returns TRUE, VOCDriverInstalled is also set to TRUE.

VOCDriver-Version

The variable VOCDriverVersion receives the driver's version number as a WORD value. So you can determine the driver's version by simply reading this variable's value, instead of calling the driver's version function.

This completes the unit's automatic initialization procedure. The advantage of this initialization is that you can determine, by checking VOCDriverInstalled in the first program line, whether your Sound Blaster card has been initialized successfully. If an error occurred, an error message appears.

Global error message

The global variable VOCErrStat allows you to detect VOCTOOL errors. This variable is assigned a specific error number when an error occurs in conjunction with VOCTOOL. If this variable contains the value 0, an error hasn't occurred.

You must determine how your program will handle these errors. Also, you must ensure that the variable is reset to zero once the error has been processed because it retains its current value until another error occurs.

The error number assigned by VOCTOOL are arranged according to number groups.

VOCTOOL error numbers

The numbers 1xx are reserved for errors that occur when the driver file is installed.

Error 100

The error number 100 is used when the CT-VOICE.DRV file couldn't be found.

Automatic driver search

At initialization, VOCTOOL.TPU automatically searches for the driver file in the directory from which the program was called. The advantage of this process is that you can copy such programs into any directory, as long as you also place a copy of the driver in the same directory.

However, this process also has a small disadvantage. A copy of CT-VOICE.DRV is located in your Turbo Pascal directory while you're developing your program. This occurs because when the unit is started from the Integrated Development Environment (IDE), it will recognize that the program call came from the Turbo directory. This makes sense when you consider that, while you're testing your program from the Integrated Development Environment (IDE), the program that's actually called is TURBO.EXE, from which your program is then started.

This is a small problem, which is eliminated once the program consists of a completed and compiled EXE file.

Once you're outside of the Turbo Pascal IDE, your program will recognize the correct search path for the driver.

Local variables

Since the newer versions of Sound Blaster set the local variable "SOUND" to indicate the SB root directory, you should prompt for this local variable.

VOCTOOL automatically does this when it doesn't find a file named CT-VOICE.DRV in its own directory.

Error 110

Error 110 occurs when an insufficient amount of memory is available for loading the driver file.

This can occur either because your system is actually out of memory or because you've forgotten to specify a memory limit within your application. In the latter case, your program will simply reserve all remaining memory for its own use, leaving no room for the driver.

This means that you'll need to specify a memory limit using the $M compiler directive. The values that will be used with this statement depend on your program.

To determine the amount of memory your program will need, use the *File/Get info...* menu command. This function provides information about the "estimated" memory requirements of your program. However, there is no way to measure this amount exactly.

Error 120

Error number 120 occurs when a loaded file isn't recognized as being a CT-VOICE.DRV file. Bytes 3 and 4 of the loaded file are compared to the characters "CT". If the values of these bytes don't match these characters, the file cannot be a driver file.

However, you can use a more complex test. The exact character sequence common to all drivers for Sound Blaster Versions 1.0 through 2.0 and the driver for Sound Blaster Pro is "CT-VOICE Creative Sound Blaster".

2xx error numbers are reserved for errors that occur while loading sample files. They are quite similar to the errors found while loading the driver file.

Error 200

Error number 200 is used when the specified VOC file couldn't be found. This can occur when an incorrect filename was specified or when the file is located in a different directory.

Error 210

Error number 210 indicates that no more memory is available for loading the sample file. This error can occur for the same reasons listed for error 110.

However, since sample files are very large, it's possible that this error will occur even if the system is set up optimally.

In this case you must either make your extended memory available or use a DOS Exec command to play back the sample file with the help of the VPLAY.EXE program.

The first solution is complicated because the driver cannot access EMS or XMS memory on its own. However, you can use the memory resident SBSIM programming interface from Creative Labs, which we briefly described in Chapter 1. When used with the correct driver, this interface is capable of managing samples in extended memory.

Although the second solution is awkward, it works quite well because VPLAY.EXE uses a buffer to play back the file directly from the floppy or hard disk.

However, if your samples aren't very long, and you don't load several larger samples into memory simultaneously, your PC's conventional memory should be sufficient.

Error 220

Error number 220 indicates that the loaded file doesn't conform to the CT-Voice format. Again, two bytes are checked to verify the file's identity. The full character sequence, which is common to all such files, is "Creative Voice File".

Error 300

Error number 300 indicates that DOS cannot make a reserved memory area available.

This error usually occurs when you try to access a memory area that you previously couldn't reserve. However, this error can also occur when DOS encounters actual memory management problems.

4xx error numbers check whether all hardware parameters are correct once the driver has been loaded successfully. The error routine uses driver function 3 to do this.

Error 400

Error 400 indicates that the driver was unable to find the Sound Blaster card. By using this error number, you can prevent your program from trying to support the Sound Blaster card on PC's that aren't equipped with the card.

Error 410

Error number 410 occurs when the port address specified in the driver file, or that you've set through driver function 1, doesn't match your card's hardware settings.

Error 420

Error number 420 indicates that the interrupt number specified in the driver file or that you've set through driver function 2, doesn't match your card's hardware settings.

By using these error messages, you can expand the unit's initialization section so the unit automatically determines the actual settings of your Sound Blaster card.

However, several system configurations can interfere with this procedure. So users should have the option of determining these settings independently.

The best way to do this is by using the fixed settings in CT-VOICE.DRV, which can be set using INST-DRV.EXE.

In the section on the driver's function you learned where this information is stored in the file. So you can easily write your own initialization program. Simply open the driver file, change the appropriate bytes, and then close the file again.

Error numbers 5xx indicate when attempts to modify the sample file playback process occur.

Error 500

Error number 500 indicates that your program has tried to interrupt a playback loop, even though a loop wasn't currently being executed.

Error 510

Error 510 occurs when your program tries to pause a sample playback, even though data wasn't currently being played back.

Error 520

Error number 520 indicates that your program has tried to continue a paused sample playback, even though a playback wasn't halted with the pause function.

Procedures & functions

In the following section we'll describe the procedures and functions that are made available to your programs through the VOCTOOL.TPU unit's interface section.

Function VOCGetVersion

This function returns a WORD value. This value contains the main and subversion number in its high and low bytes.

Procedure VOCSetPort

With this procedure you can change the port address, which the driver must use for its initialization, before the driver is actually installed.

Since the driver's automatic initialization prevents you from calling this function before the driver is initialized, this procedure is included only to complete the palette of functions.

If you want to use this function, you must write your own appropriate initialization procedure.

Procedure VOCSetIRQ

The VOCSetIRQ procedure allows you to change the desired interrupt number before the driver is initialized.

This procedure is also included only to complete the range of functions. The conditions that apply to VOCSetPort also apply to this procedure.

Function VOCInitDriver

This function is needed only in certain instances because it's automatically called by the unit's initialization section. However, you may want to switch off the Sound Blaster card completely and initialize it again later in your program. This is why the function is included in the interface section.

If you perform the initialization later by using VOCInitDriver, you'll also be able to use VOCSetPort and VOCSetIRQ at that time.

VOCInitDriver has been performed successfully only when the function value of the corresponding driver function is zero. All other values represent an error, which can be determined by reading the value of the VOCErrStat variable.

Procedure VOCDeInstallDriver

The VOCDeInstallDriver procedure switches off the Sound Blaster card's output and removes the driver from memory. The memory area occupied by the driver is then freed.

This procedure is called by the unit's new ExitProc and is needed only in special cases.

Procedure PrintVOCErrMessage

This procedure prints the current error on your screen as text. Since the procedure simply uses a Write statement, you can easily incorporate this procedure into your own program's error message system or modify the procedure according to your own preferences.

Function VOCGetBuffer

VOCGetBuffer performs all the tasks that are needed for a sample file to be loaded into memory. Simply specify a pointer variable as well as the name of the file you want to load.

This function then tries to open the file and reserve the required memory. After this the data are loaded into memory. The file's contents may, of course, exceed the size of a segment or 64K.

Once these steps have been performed successfully, the function value returns TRUE and the pointer variable points to the beginning of the reserved memory area.

If FALSE is returned, you must evaluate the error numbers and respond accordingly.

Function VOCFreeBuffer

This function frees the memory area reserved by the VOCGetBuffer function. The only parameter that's required is the pointer variable identifying the desired memory area.

Error 300 can occur only during this procedure. As we mentioned, this error occurs only when you try to free a memory area that hasn't been reserved. This can happen, for example, when an incorrect pointer variable is specified. Depending on whether an error occurs, the returned function value is either TRUE or FALSE.

Procedure VOCSetSpeaker

This function switches the speaker output of your Sound Blaster card on or off. If TRUE is passed as a parameter, the speaker output is switched on, and with FALSE the output is switched off.

Once the driver is successfully initialized, the speaker is automatically switched on. The speaker is switched off when the driver is de-installed.

Procedure VOCOutput

This procedure allows a sample located in memory to be played over your Sound Blaster card. The only required parameter is the pointer variable identifying the data that must be played back.

Control returns to the program immediately after the procedure is started because the DMA enables the sample playback to occur at the same time as other program executions.

In your program, you have the option of checking the VOCStatusWord variable. As long as this variable contains a value that doesn't equal zero, data is still being sent to the DAC

(or from the ADC to memory, if you also implement the recording of samples as an additional Pascal procedure).

When a marker block is reached during the playback of a sound sample, its contents are copied to the VOCStatusWord variable. This allows you to check which portion of the sample is currently being played so you can synchronize the playback with other events.

Procedure VOCOutputLoop

The VOCOutputLoop procedure is identical to VOCOutput, although switching the speaker on when it's already activated leads to unpleasant cracking sounds on some Sound Blaster cards. This is especially annoying when a sound must be played repeatedly (i.e., when a loop is being played).

Therefore, this procedure doesn't switch on the speaker. So you must verify whether the speaker is switched on when you call VOCOutputLoop.

ProcedureVOCPause/Procedure VOCContinue

VOCPause allows you to pause the playback of a sample, which can later be continued using VOCContinue.

VOCPause sets the VOCPaused global variable to TRUE, if a sample was actually being played back. This variable, therefore, allows you to check whether a sample playback is currently paused.

VOCContinue resets VOCPaused to FALSE, if it continued the sample playback successfully.

ProcedureVOCStop

This procedure terminates the playback of a sample. This is important when you want to play back a new sample before the entire first sample has been played.

Never try to start the playback of a sample while another sample is being played back because this leads to a system crash.

Procedure VOCBreakLoop

This procedure allows you to interrupt a loop specified by the CT-Voice format within a sample's data.

To interrupt such a loop, pass either the value 0 or 1 as the procedure parameter.

This parameter determines whether the loop will be interrupted only once the present pass has been completed or whether it will be interrupted immediately. To make this a little easier to remember, the constants VOCBreakEnd and VOCBreakNow have been assigned the corresponding values.

Room for improvement

The functions used in the VOCTOOL unit cover all the functions offered by the driver, except for its own sample playback. However, this function is performed so well by so many other functions that you'll need it only for special applications.

However, there are many other ways you can expand this unit. One interesting addition would be reading the environment variables to determine the correct sound parameters.

The "SOUND" variable has already been considered, but you could also consider using the "BLASTER" variable to determine the port address and IRQ number.

Also, saving sounds, which you've recorded, as files is another option that's missing. However, this is an easy task to perform; simply reverse the procedure performed by VOCGetBuffer.

Driver function 13, the user-defined function, is another function that hasn't been used in VOCTOOL.TPU because it isn't needed. The procedure is tailored to each specific use so you must make your own additions here.

As you can see, this unit can be expanded and improved in various ways. You could also write your own VOC file editor by using your knowledge of the structure of the CT-Voice format. However, the Creative Labs Voice Editor, as well as other shareware products, don't have many limitations.

If you expand the VOCTOOL.PAS unit, feel free to change the VOCToolVersion constant defined at the beginning of the unit. Currently it contains the string "v1.6", which indicates that this is Version 1.6 of this particular unit.

If you update this version number each time you modify the unit, you'll always know with which version of your modified unit you're currently working.

```
UNIT VOCTOOL;
{
 *******************************************************************
 * Unit for Sound Blaster card control in Turbo Pascal 6.0, using   *
 *          the CT-VOICE.DRV driver.                                *
 *******************************************************************
 *                          (C) 1992 Abacus                         *
 *                        Author : Axel Stolz                       *
 *******************************************************************
}
INTERFACE

TYPE
   VOCFileTyp = File;

CONST
   VOCToolVersion   = 'v1.5'; { Version number for this VOCTOOL unit }
   VOCBreakEnd      = 0;      { Constant for loop break              }
   VOCBreakNow      = 1;      { Constant for loop break              }

VAR
   VOCStatusWord        : WORD;     { Sound Blaster status variable }
   VOCErrStat           : WORD;     { Driver error number variable  }
   VOCFileHeader        : STRING;   { CT format header variable     }
   VOCFileHeaderLength  : BYTE;     { CT format header length       }
   VOCPaused            : BOOLEAN;  { Flag for VoiceStatus pause    }
   VOCDriverInstalled   : BOOLEAN;  { Flag, if driver installed     }
   VOCDriverVersion     : WORD;     { Driver version number         }
   VOCPtrToDriver       : Pointer;  { Pointer to driver in memory   }
   OldExitProc          : Pointer;  { Pointer to old unit ExitProc  }


PROCEDURE PrintVOCErrMessage;
FUNCTION  VOCGetBuffer(VAR VoiceBuff : Pointer; Voicefile : STRING):BOOLEAN;
FUNCTION  VOCFreeBuffer(VAR VoiceBuff : Pointer):BOOLEAN;
FUNCTION  VOCGetVersion:WORD;
PROCEDURE VOCSetPort(PortNumber : WORD);
PROCEDURE VOCSetIRQ(IRQNumber : WORD);
FUNCTION  VOCInitDriver:BOOLEAN;
PROCEDURE VOCDeInstallDriver;
PROCEDURE VOCSetSpeaker(OnOff:BOOLEAN);
PROCEDURE VOCOutput(BufferAddress : Pointer);
PROCEDURE VOCOutputLoop (BufferAddress : Pointer);
PROCEDURE VOCStop;
PROCEDURE VOCPause;
PROCEDURE VOCContinue;
PROCEDURE VOCBreakLoop(BreakMode : WORD);

IMPLEMENTATION
```

```
USES DOS,Crt;

TYPE
   TypeCastType = ARRAY [0..6000] of Char;

VAR
   Regs : Registers;

PROCEDURE PrintVOCErrMessage;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Returns SB error on the screen as text, without changing
 *           error status.
}
BEGIN
   CASE VOCErrStat OF
      100 : Write(' Driver file CT-VOICE.DRV not found ');
      110 : Write(' No memory available for driver file ');
      120 : Write(' False driver file ');

      200 : Write(' VOC file not found ');
      210 : Write(' No memory available for driver file ');
      220 : Write(' File not in VOC format ');

      300 : Write(' Memory allocation error occurred ');

      400 : Write(' No sound blaster card found ');
      410 : Write(' False port address used ');
      420 : Write(' False interrupt used ');

      500 : Write(' No loop in process ');
      510 : Write(' No sample for output ');
      520 : Write(' No sample available ');
      END;
   END;

FUNCTION Exists (Filename : STRING):BOOLEAN;
{
 * INPUT   : Filename as string
 * OUTPUT  : TRUE if file is available, FALSE if not
 * PURPOSE : Checks for availability of a file then returns a Boolean
             expression.
}
VAR
   F : File;
BEGIN
   Assign(F,Filename);
{$I-}
   Reset(F);
   Close(F);
{$I+}
   Exists := (IoResult = 0) AND (Filename <> '');
```

```
   END;

PROCEDURE AllocateMem (VAR Pt : Pointer; Size : LongInt);
{
 * INPUT   : Buffer variable as pointer, buffer size as LongInt
 * OUTPUT  : Pointer to buffer in variable or NIL
 * PURPOSE : Reserves as many bytes as Size allows, then moves pointer
             in the Pt variable. If not enough memory is available, Pt = NIL.
}
VAR
   SizeIntern : WORD;     { Pointer size for internal calculation }
BEGIN
   Inc(Size,15);                 { Increment buffer size by 15... }
   SizeIntern := (Size shr 4);   { then divide by 16.             }
   Regs.AH := $48;               { Place DOS function $48 in AH   }
   Regs.BX := SizeIntern;        { Place internal size in BX      }
   MsDos(Regs);                  { Reserve memory                 }
   IF (Regs.BX <> SizeIntern) THEN Pt := NIL
   ELSE Pt := Ptr(Regs.AX,0);
   END;

FUNCTION  CheckFreeMem (VAR VoiceBuff : Pointer; VoiceSize :
LongInt):BOOLEAN;
{
 * INPUT   : Buffer variable as pointer, size as LongInt
 * OUTPUT  : Pointer to buffer, TRUE/FALSE, after AllocateMem
 * PURPOSE : Checks for sufficient memory to store a VOC file.
}
BEGIN
   AllocateMem(VoiceBuff,VoiceSize);
   CheckFreeMem := VoiceBuff <> NIL;
   END;

FUNCTION  VOCGetBuffer (VAR VoiceBuff : Pointer; Voicefile : STRING):BOOLEAN;
{
 * INPUT   : Buffer variable as pointer, file name as string
 * OUTPUT  : Pointer to buffer with VOC data, TRUE/FALSE
 * PURPOSE : Loads a file into memory and returns TRUE if file loaded
             successfully, and FALSE if not.
}
VAR
   SampleSize : LongInt;
   FPresent   : BOOLEAN;
   VFile      : VOCFileTyp;
   Segs       : WORD;
   Read       : WORD;

BEGIN
   FPresent := Exists(VoiceFile);

{ VOC file could not be found }
   IF Not(FPresent) THEN BEGIN
      VOCGetBuffer := FALSE;
      VOCErrStat   := 200;
```

```
      EXIT
      END;

   Assign(VFile,Voicefile);
   Reset(VFile,1);
   SampleSize := Filesize(VFile);
   AllocateMem(VoiceBuff,SampleSize);

{ Insufficient memory for the VOC file }
   IF (VoiceBuff = NIL) THEN BEGIN
      Close(VFile);
      VOCGetBuffer := FALSE;
      VOCErrStat   := 210;
      EXIT;
      END;

   Segs := 0;
   REPEAT

Blockread(VFile,Ptr(seg(VoiceBuff^)+4096*Segs,Ofs(VoiceBuff^))^,$FFFF,Read);
      Inc(Segs);
      UNTIL Read = 0;
   Close(VFile);

{ File not in VOC format }
   IF (TypeCastType(VoiceBuff^)[0]<>'C') OR
      (TypeCastType(VoiceBuff^)[1]<>'r') THEN BEGIN
      VOCGetBuffer := FALSE;
      VOCErrStat := 220;
      EXIT;
      END;

{ Load successful }
   VOCGetBuffer := TRUE;
   VOCErrStat   := 0;

{ Read header length from file }
   VOCFileHeaderLength := Ord(TypeCastType(VoiceBuff^)[20]);
   END;

FUNCTION VOCFreeBuffer (VAR VoiceBuff : Pointer):BOOLEAN;
{
 * INPUT   : Buffer pointer
 * OUTPUT  : None
 * PURPOSE : Frees memory allocated for VOC data.
}
BEGIN
   Regs.AH := $49;                { Place function $49 in AH   }
   Regs.ES := seg(VoiceBuff^);    { Place memory segment in ES }
   MsDos(Regs);                   { Release memory             }
   VOCFreeBuffer := TRUE;
   IF (Regs.AX = 7) OR (Regs.AX = 9) THEN BEGIN
      VOCFreeBuffer := FALSE;
      VOCErrStat := 300         { Release causes a DOS error }
```

```
      END;
   END;

FUNCTION VOCGetVersion:WORD;
{
 * INPUT   : None
 * OUTPUT  : Driver version number
 * PURPOSE : Returns driver version number.
}
VAR
   VDummy : WORD;
BEGIN
   ASM
      MOV       BX,0
      CALL      VOCPtrToDriver
      MOV       VDummy, AX
      END;
   VOCGetVersion := VDummy;
   END;

PROCEDURE VOCSetPort(PortNumber : WORD);
{
 * INPUT   : Port address number
 * OUTPUT  : None
 * PURPOSE : Specifies port address before initialization.
}
BEGIN
   ASM
      MOV    BX,1
      MOV    AX,PortNumber
      CALL   VOCPtrToDriver
      END;
   END;

PROCEDURE VOCSetIRQ(IRQNumber : WORD);
{
 * INPUT   : Interrupt number
 * OUTPUT  : None
 * PURPOSE : Specifies interrupt number before initialization.
}
BEGIN
   ASM
      MOV    BX,2
      MOV    AX,IRQNumber
      CALL   VOCPtrToDriver
      END;
   END;

FUNCTION  VOCInitDriver: BOOLEAN;
{
 * INPUT   : None
 * OUTPUT  : Error message number, and initialization result
 * PURPOSE : Initializes driver software.
```

```
VAR
   Out, VSeg, VOfs : WORD;
   F   : File;
   Drivername,
   Pdir         : DirStr;
   Pnam         : NameStr;
   Pext         : ExtStr;

BEGIN
{ Search path for CT-VOICE.DRV driver }
   Pdir := ParamStr(0);
   Fsplit(ParamStr(0),Pdir,Pnam,Pext);
   Drivername := Pdir+'CT-VOICE.DRV';

   VOCInitDriver := TRUE;

{ Driver file not found }
   IF Not Exists(Drivername) THEN BEGIN
      VOCInitDriver := FALSE;
      VOCErrStat    := 100;
      EXIT;
      END;

{ Load driver }
   Assign(F,Drivername);
   Reset(F,1);
   AllocateMem(VOCPtrToDriver,Filesize(F));

{ No memory can be allocated for the driver }
   IF VOCPtrToDriver = NIL THEN BEGIN
      VOCInitDriver := FALSE;
      VOCErrStat    := 110;
      EXIT;
      END;

   Blockread(F,VOCPtrToDriver^,Filesize(F));
   Close(F);

{ Driver file doesn't begin with "CT" - false driver }
   IF (TypeCastType(VOCPtrToDriver^)[3]<>'C') OR
      (TypeCastType(VOCPtrToDriver^)[4]<>'T') THEN BEGIN
         VOCInitDriver := FALSE;
         VOCErrStat    := 120;
         EXIT;
         END;

{ Get version number and pass to global variable }
   VOCDriverVersion := VOCGetVersion;

{ Start driver }
   Vseg := Seg(VOCStatusWord);
   VOfs := Ofs(VOCStatusWord);
   ASM
      MOV       BX,3
```

```
      CALL      VOCPtrToDriver
      MOV       Out,AX
      MOV       BX,5
      MOV       ES,VSeg
      MOV       DI,VOfs
      CALL      VOCPtrToDriver
      END;

{ No Sound Blaster card found  }
   IF Out = 1 THEN BEGIN
      VOCInitDriver := FALSE;
      VOCErrStat    := 400;
      EXIT;
      END;

{ False port address used      }
   IF Out = 2 THEN BEGIN
      VOCInitDriver := FALSE;
      VOCErrStat    := 410;
      EXIT;
      END;

{ False interrupt used         }
   IF Out = 3 THEN BEGIN
      VOCInitDriver := FALSE;
      VOCErrStat    := 420;
      EXIT;
      END;
   END;

PROCEDURE VOCDeInstallDriver;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Disables driver and releases memory.
}
VAR
   Check : BOOLEAN;
BEGIN
   IF VOCDriverInstalled THEN
   ASM
      MOV       BX,9
      CALL      VOCPtrToDriver
      END;
   Check := VOCFreeBuffer(VOCPtrToDriver);
   END;

PROCEDURE VOCSetSpeaker(OnOff:BOOLEAN);
{
 * INPUT   : TRUE=Speaker on, FALSE=Speaker off
 * OUTPUT  : None
 * PURPOSE : Sound Blaster output status.
}
VAR
```

```
   Switch : BYTE;
BEGIN
   Switch := Ord(OnOff) AND $01;
   ASM
      MOV       BX,4
      MOV       AL,Switch
      CALL      VOCPtrToDriver
      END;
   END;

PROCEDURE VOCOutput (BufferAddress : Pointer);
{
 * INPUT   : Pointer to sample data
 * OUTPUT  : None
 * PURPOSE : Plays sample.
}
VAR
   VSeg, VOfs : WORD;
BEGIN
   VOCSetSpeaker(TRUE);
   VSeg := Seg(BufferAddress^);
   VOfs := Ofs(BufferAddress^)+VOCFileHeaderLength;
   ASM
      MOV       BX,6
      MOV       ES,VSeg
      MOV       DI,VOfs
      CALL      VOCPtrToDriver
      END;
   END;

PROCEDURE VOCOutputLoop (BufferAddress : Pointer);
{
 *    Different from VOCOutput :
 *    Speaker does not switch on with every sample output, so a
 *    crackling noise may occur with some Sound Blaster cards.
}
VAR
   VSeg, VOfs : WORD;
BEGIN
   VSeg := Seg(BufferAddress^);
   VOfs := Ofs(BufferAddress^)+VOCFileHeaderLength;
   ASM
      MOV       BX,6
      MOV       ES,VSeg
      MOV       DI,VOfs
      CALL      VOCPtrToDriver
      END;
   END;

PROCEDURE VOCStop;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Stops a sample.
```

```
}
BEGIN
   ASM
      MOV      BX,8
      CALL     VOCPtrToDriver
      END;
   END;

PROCEDURE VOCPause;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Pauses a sample.
}
VAR
   Switch : WORD;
BEGIN
   VOCPaused := TRUE;
   ASM
      MOV      BX,10
      CALL     VOCPtrToDriver
      MOV      Switch,AX
      END;
   IF (Switch = 1) THEN BEGIN
      VOCPaused := FALSE;
      VOCErrStat := 510;
      END;
   END;

PROCEDURE VOCContinue;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Continues a paused sample.
}
VAR
   Switch : WORD;
BEGIN
   ASM
      MOV      BX,11
      CALL     VOCPtrToDriver
      MOV      Switch,AX
      END;
   IF (Switch = 1) THEN BEGIN
      VOCPaused := FALSE;
      VOCErrStat := 520;
      END;
   END;

PROCEDURE VOCBreakLoop(BreakMode : WORD);
{
 * INPUT   : Break mode
 * OUTPUT  : None
 * PURPOSE : Breaks a sample loop.
```

```
}
BEGIN
   ASM
      MOV       BX,12
      MOV       AX,BreakMode
      CALL      VOCPtrToDriver
      MOV       BreakMode,AX
      END;
   IF (BreakMode = 1) THEN VOCErrStat := 500;
   END;

{$F+}
PROCEDURE VoiceToolsExitProc;
{$F-}
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : De-installs voice driver.
}
BEGIN
   VOCDeInstallDriver;
   ExitProc := OldExitProc;
   END;

BEGIN
{
 * The following statements execute automatically, as soon as the
 * unit is linked to a program, and the program starts.
}
{ Replaces old ExitProc with new one from Tool unit }
   OldExitProc := ExitProc;
   ExitProc := @VoiceToolsExitProc;
{ Initialize values }
   VOCStatusWord := 0;
   VOCErrStat    := 0;
   VOCPaused     := FALSE;
   VOCFileHeaderLength := $1A;
   VOCFileHeader :=
      'Creative Voice File'+#$1A+#$1A+#$00+#$0A+#$01+#$29+#$11+#$01;
{
 * After installation, VOCDriverInstalled contains either TRUE or FALSE,
 * depending on whether driver installation was successful or not.
}
   VOCDriverInstalled := VOCInitDriver;
   END.
```

VTTEST

The VTTEST.PAS program demonstrates how the basic functions of the VOCTOOL.TPU unit can be used.

Running VTTEST.EXE

Make sure that the CT-VOICE.DRV driver is either in the same directory as VTTEST.EXE, or accessible. Make sure the

\DEMO.VOC file is in the same directory as VTTEST.EXE. Run VTTEST.EXE.

Setting the memory range

As we mentioned, first you must use the $M compiler directive to specify the upper memory limit for your program. In our example, these values are 16000 bytes for the stack and 0 to 50000 bytes for the heap.

The 50000-byte heap limit should be more than enough. As long as memory problems don't occur when you try to load sample files, you may want to grant your program a few extra bytes of heap memory. By doing this, you can avoid determining exactly how much heap your program needs.

You'll receive an approximate figure through the *File/ Get info...* menu command, indicated by the "Code Size" and "Data Size" values.

However, if you're using other pointer variables in addition to those for the sound sample data, the compiler won't recognize how much memory is needed for the data represented by these pointers.

In the next step, the USES statement is used to link the Crt unit, which is responsible for the screen functions, and the VOCTOOL unit. This gives you access to the procedures and functions offered by these units, and the unit's initialization section is executed at the start of your program.

TextNumError procedure

If an error occurs, the TextNumError procedure in our program example is responsible for displaying the error number and the corresponding error text. The program is then interrupted by the HALT statement and an ErrorLevel is returned to DOS, indicating which error number caused the program to be interrupted.

This allows you, for example within a batch file, to determine why the program was halted and to correct the error. If the program was executed without any errors, the error level is zero; otherwise it contains the VOCErrStat error number.

Main program

The first few lines of the main program make the preparations needed for playing back a sound sample that weren't resolved by the VOCTOOL.TPU unit.

Flexible error processing

The first line checks whether the VOCDriverInstalled global variable contains the value TRUE. If it doesn't, the TextNumError

procedure is called. The program will be halted with ErrorLevel 100 to 120 or 400 to 420, depending on the cause of the error.

Perhaps your program supports the PC speaker in addition to the Sound Blaster card. In this case, your program shouldn't be halted because of the failed driver installation. You may want to set a flag at this point, indicating, to the rest of your program, that the Sound Blaster card isn't available. This demonstrates that the way in which errors are processed can be adapted according to the needs of your particular program.

Loading a VOC File

In the second program line, VOCGetBuffer is used to load the sound sample file DEMO.VOC into memory. The Sound variable is used as a pointer to the sample data. Depending on whether this file is loaded successfully, the function value TRUE or FALSE is assigned to the Check variable.

If Check contains FALSE, the program is halted, along with the corresponding message from the TextNumError procedure. Otherwise the main portion of the program is executed.

In the main portion, the driver version number is displayed. Then the user is asked to press S (for single play) or M (for multiple play).

Depending on the user's entry, the CASE statement then branches either to an "S" portion or an "M" portion.

A single sample playback

In the "S" portion, first a help text is displayed, informing the user that the playback of the sound can be interrupted by pressing any key. Then the playback of the sample is started by using the VOCOutput procedure.

Since the procedure returns immediately after the playback has been started, the main program proceeds to a Repeat/Until... wait loop. This loop checks whether the user has pressed a key to interrupt the playback or whether the playback has been completed. The VOCStatusWord variable is checked to determine this. If a keystroke is detected, the sample's playback is stopped.

Multiple playbacks

In the "M" portion of the CASE statement, the sound is played back repeatedly until the user presses the Esc key. The VOCOutputLoop procedure is used to do this.

Before using VOCOutputLoop, ensure that the Sound Blaster speaker is switched on. In this program the speaker is

automatically switched on when the driver is initialized successfully.

The Repeat/Until... wait loop in this portion checks whether a key has been pressed or whether the playback is complete.

If a key has been pressed, its ASCII code is assigned to the variable Ch. If this value isn't equal to 27 (the ASCII code for ESC), the outer Repeat/Until... loop is started and the sample is replayed.

These sound loops are particularly useful for adding music to graphic applications. With the Voice Editor you can easily edit short digitized musical sequences so it becomes impossible to tell where the sample ends and where it begins again.

By doing this, you can create a continuous musical loop without a large amount of memory. The ARRANGER.PAS program also uses this procedure.

At the end of the program, the VOCFreeBuffer procedure is used to free the memory area that was occupied by the sound sample data. Then the program ends.

In the background, the ExitProc of the VOCTOOL.TPU unit also switches off the Sound Blaster card and removes the driver from memory.

The VTTEST.PAS program shows you some ways you can incorporate sample playbacks into your own programs.

```
PROGRAM VToolTest;
{
 ************************************************************************
 *  Demonstration program for VOCTOOL unit, written in Turbo Pascal 6.0  *
 ************************************************************************
 *                           (C) 1992 Abacus                             *
 *                          Author : Axel Stolz                          *
 ************************************************************************
 * Memory limits must be allocated by the program, or the program will   *
 * use all available memory, leaving no additional memory for the driver *
 * and sample data. The memory parameter must be passed to the           *
 * main program.                                                         *
 ************************************************************************
}

{$M 16000,0,50000}

USES Crt,Voctool;
```

```
VAR
   Sound : Pointer; { Pointer to sample data in memory }
   Check : BOOLEAN; { Flag for Boolean check           }
   Ch    : CHAR;    { Character buffer for user input  }

PROCEDURE TextNumError;
{
 * INPUT   : None; data comes from the VOCErrStat global variable
 * OUTPUT  : None
 * PURPOSE : Displays SB error on the screen as text, including the
             error number. Program then ends at the error level
             corresponding to the error number.
}
BEGIN
   Write(' Error #',VOCErrStat:3,' =');
   PrintVOCErrMessage;
   WriteLn;
   HALT(VOCErrStat);
   END;

BEGIN
  ClrScr;

{ Driver not initialized }
  IF Not(VOCDriverInstalled) THEN TextNumError;

{ Loads DEMO.VOC file into memory }
  Check := VOCGetBuffer(Sound,'DEMO.VOC');

{ VOC file could not be loaded }
  IF Not(Check) THEN TextNumError;

{ Main loop }
  Write('CT-Voice Driver Version : ');
  WriteLn(Hi(VOCDriverVersion),'.',Lo(VOCDriverVersion));
  WriteLn('(S)ingle play or (M)ultiple play?');
  Write('Press a key : '); Ch := ReadKey;WriteLn;WriteLn;
  CASE UpCase(Ch) OF
   'S' : BEGIN
           Write('Press a key to stop the sound...');
           VOCOutput(Sound);
           REPEAT UNTIL KeyPressed OR (VOCStatusWord = 0);
           IF KeyPressed THEN VOCStop;
           END;
   'M' : BEGIN
           Ch := #0;
           Write('Press <ESC> to cancel...');
           REPEAT
              VOCOutputLoop(Sound);
              REPEAT UNTIL KeyPressed OR (VOCStatusWord = 0);
              IF KeyPressed THEN Ch := ReadKey;
              UNTIL Ch = #27;
           VOCStop;
```

```
            END;
   END;

{ Free VOC file memory }
  Check := VOCFreeBuffer(Sound);
  IF Not(Check) THEN TextNumError;
  END.
```

ARRANGER

You've probably already been surprised by the high quality of digitized Sound Blaster samples that are available. For example, one programmer created an elaborate sample from a tune performed by the band Art of Noise.

You can edit digitized scores in the same way using the Voice Editor. The ARRANGER.PAS program provides a simple way of joining the individually edited sound segments.

The new variable type PartType is declared. This variable is a record with three components. The first component is a pointer that will point to the sound sample data stored in memory. The second component is the name that will be assigned to this sound sample segment within the arrangement and the third is the name of the file containing the sample data.

The global constant MaxParts specifies how many individual sample segments will make up the arrangement. Our example has five segments.

The OK.VOC and LETSGO.VOC files are digitized speech samples. The INTRO.VOC file contains the beginning of a musical piece. The RIFF.VOC file contains the song's main loop, which can be repeated as often as desired. The END.VOC file contains an appropriate end segment for the riff. The arrangement concludes with the OK.VOC sample.

The Parts constant contains all the necessary information about these five files. Since the pointers for the files don't point to any specific memory location yet, they are set to NIL.

The pattern name of the first piece in the arrangement, for example, is OK. The name OK.VOC is assigned to this pattern name. The file INTRO.VOC receives the pattern name I1. The same procedure is used for the other pattern names.

Pattern sequence

The DemoArrangement constant is then assigned the information specifying in which sequence these patterns should be played

back. The character sequence "OKLGI1R1R1E1OK" specifies that OK.VOC and LETSGO.VOC should be played back first.

Then the musical piece begins. First the intro is played (i.e., the data represented by the pattern name "I1"). Next the middle portion of the piece is played. This section is played twice, and if it doesn't get too boring, you may want to add more "R1"s at the appropriate location to repeat the segment a few more times.

Then END.VOC is played and OK.VOC is replayed.

With this pattern list you can determine the number of times and the sequence in which the loaded sample data are played back.

Expansion

You can modify the ARRANGER.PAS program so these settings aren't specified as constants at the beginning of the code. Instead, the program could read the data from an ASCII file.

If you modified the program in this way, it would be possible to play various arrangements with the ARRANGER.PAS program.

By using more than one string variable, it's also possible to increase the maximum number of patterns that can be played consecutively. A single Pascal string is limited to 255 characters, allowing up to 127 pattern names to be stored in one string variable.

You'll probably think of other useful changes. Once you've learned how to edit sound samples so they sound natural when played consecutively, you can begin to create larger pieces of music.

```
Program Arranger;
{
 **************************************************************************
 * Demonstration program for VOCTOOL unit,  written in Turbo Pascal 6.0   *
 **************************************************************************
 *                          (C) 1992  Abacus                              *
 *                         Author : Axel Stolz                            *
 **************************************************************************
 * This program lets the user arrange sample files in any sequence so     *
 * that samples can be played as a group of musical excerpts or noises.   *
 * ARRANGER allows 127 enabled sample patterns, stated in a string up to *
 * 255 characters in length. Including additional strings enables         *
 * longer pattern sequences.                                              *
 **************************************************************************
}

{$M 16000,0,50000}

USES Crt, Voctool;
```

```
TYPE                                 { An arrangement section consists of: }
  PartType = RECORD
         PPoint : Pointer;           { A pointer to the sample file, a      }
         PName  : STRING[2];         { pattern name for the sample and a    }
         PFile  : STRING[12];        { file name for the sample file        }
         END;

CONST                                     { Example of a brief arrangement }
   MaxParts = 5;
   Parts : ARRAY[1..MaxParts] OF PartType =
               ((PPoint: NIL; PName: 'OK'; PFile : 'OK.VOC'),
                (PPoint: NIL; PName: 'LG'; PFile : 'LETSGO.VOC'),
                (PPoint: NIL; PName: 'I1'; PFile : 'INTRO.VOC'),
                (PPoint: NIL; PName: 'R1'; PFile : 'RIFF.VOC'),
                (PPoint: NIL; PName: 'E1'; PFile : 'END.VOC')
               );
   DemoArrangement  = 'OKLGI1R1R1E1OK';

PROCEDURE TextNumError;
{
 * INPUT   : None - data comes from the VoiceErrStat global variable
 * RETURNS : None
 * PURPOSE : Displays the SB error on the screen as text (including the
             error number), then ends the program with the error level
             corresponding to the error number.
}
BEGIN
   Write(' Error #',VOCErrStat:3,': ');
   PrintVOCErrMessage;
   WriteLn;
   HALT(VOCErrStat);
   END;

PROCEDURE LoadParts;
{
 * INPUT   : None
 * RETURNS : None
 * PURPOSE : Sample files are loaded into memory, based on the contents
             of the Parts variable.
}
VAR
   PCount   : BYTE;
   LoadFlag : BOOLEAN;
BEGIN
   FOR PCount := 1 TO MaxParts DO BEGIN
     LoadFlag := VOCGetBuffer(Parts[PCount].PPoint,Parts[PCount].PFile);
     IF Not(LoadFlag) THEN BEGIN
        WriteLn(Parts[PCount].PFile);
        TextNumError;
        END;
     END;
   END;

Procedure PlayParts(PartArrangement : STRING);
```

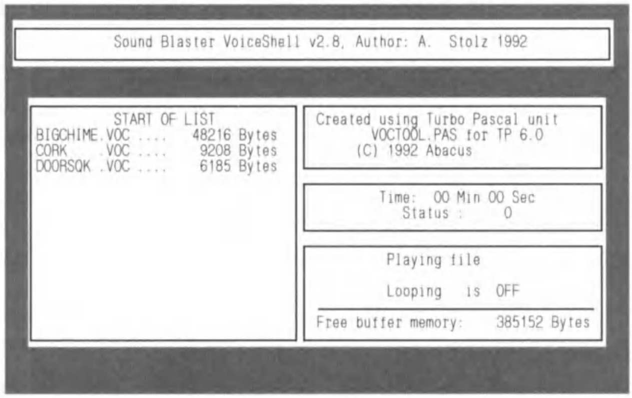
```
{
 * INPUT   : STRING containing sample names
 * RETURNS : None
 * PURPOSE : Plays the sample sequence stored in the PartArrangement
             variable. No confirmation asks whether the sequence is
             valid or not.
}
VAR
   MaxPattern : WORD;                  { Number of patterns        }
   VOCArrange : ARRAY[1..127] OF Byte; { Buffer for Sample numbers }
   ActCount,
   PartCount  : WORD;                  { Counter variables         }
   ActPattern : STRING[2];             { Buffer for pattern name   }
BEGIN
{ Determines how many patterns are in the PartArrangement string }
   MaxPattern    := (Length(PartArrangement) div 2);
{ Specifies a sample number for pattern, then stores it }
   FOR ActCount := 1 TO MaxPattern DO BEGIN
      ActPattern := Copy(PartArrangement,ActCount*2-1,2);
      PartCount := 1;
      WHILE PartCount <= MaxParts DO BEGIN
         IF ActPattern = Parts[PartCount].PName THEN BEGIN
            VOCArrange[ActCount] := PartCount;
            PartCount:= MaxParts;
            END;
         INC(PartCount);
         END;
      END;
{ Lists all samples in a box on the screen }
   WriteLn('┌──────────────┬───────────┬────────────────────┐');
   FOR PartCount := 1 TO MaxPattern DO BEGIN
      Write  ('│ Pattern # ',PartCount:3);
      Write  ('│ Name : ',Parts[VOCArrange[PartCount]].PName:2,' ');
      Write  ('│ File : ',Parts[VOCArrange[PartCount]].PFile:12,' ');
      WriteLn('│');
      VOCOutputLoop(Parts[VOCArrange[PartCount]].PPoint);
      REPEAT UNTIL VOCStatusWord = 0;
      END;
   WriteLn('└──────────────┴───────────┴────────────────────┘');
   END;

BEGIN
   ClrScr;
   IF Not(VOCDriverInstalled) THEN TextNumError;
   LoadParts;
   PlayParts(DemoArrangement);
   END.
```

VOCSHELL

Besides the programs and the VOCTOOL.TPU unit, the companion diskette also contains a program called VOCSHELL.EXE. Copy

the CT-VOICE.DRV driver to the same directory as VOCSHELL, then run VOCSHELL.EXE.

This program, written using the VOCTOOL.TPU unit, provides an interface from which you can easily play back VOC files.



VOCSHELL interface

In the left half of the work window, you'll see a file list from which you can select any desired VOC file. Press Enter or P to activate the playback of that file.

The maximum file size that VOCSHELL is capable of playing back at one time is displayed on the right side of the window as "Free buffer memory".

Once a playback is started successfully, VocShell also displays the current value of VOCStatusWord as well as the time elapsed since the start of the playback on the right side of the window.

You can interrupt the playback of a VOC file at any time by simply pressing Esc.

Test ARRANGER samples

When you press L you can switch the playback loop function on or off. The current status of this function is also displayed on the right side of the window.

This option is particularly useful when you want to determine whether you've edited a sample correctly so it can be used in the ARRANGER, for example. The selected sample is then repeated continually until you press Esc.

You can also use the file list to access other directories or to look for additional VOC files.

However, with VOCSHELL you can play sample files without first selecting the file through the VOCSHELL interface.

You can simply specify the desired filename as a command line parameter and add the /S switch to suppress all screen output. If you also want to use the loop function, add the /L switch.

By calling VOCSHELL with the /H switch, you'll display a help text explaining the required syntax.

The VOCDEMO.BAT batch file will show you how you can use VOCSHELL with its different parameters.

VOCSHELL can be particularly useful if you're searching your hard disk for a particular VOC file or even if you just want to demonstrate a few good sound samples.

5.2.4 VOC programming with Borland C++ 3.1

The companion diskette also contains a C module for programming with the CT-VOICE.DRV driver. This should make the digital channel accessible to C programmers.

The structure of this module is almost identical to the structure of the Pascal unit, which we described in the previous section. However, we'll describe the structure of the C module in this section.

If you've already read the Pascal section, the information in this section may sound familiar. However, it's important that you read this section because the many details are important.

Integrated assembler

VOCTOOL.H contains the complete interface between C and the driver. Even though this file was created using Borland C++ 3.1, it should be compatible with every C compiler that has an integrated assembler. If you don't have access to such a C compiler, you must create an assembler module that's tied to your project with the linker.

The .H extension of VOCTOOL.H indicates a header file. So adding program code to this file goes against C convention. However, once you change this extension to .C and add this file to your program project, you'll be back in C style.

First the #define directive is used to create several abbreviations that will be needed within the module. These are abbreviations for the values TRUE and FALSE, as well as for the data types WORD, BYTE, and VOCPTR.

Then three constants, which will also be required by the module, are defined.

voctool_version

The voctool_version constant contains a character string specifying the version of the module. If you change the module at any point, increase the version number accordingly.

voc_breakend / voc_breaknow

The two constants voc_breakend and voc_breaknow can be used when interrupting CT-format playback loops. voc_breaknow interrupts the loop immediately, while voc_breakend allows the current pass to be completed before the loop is stopped.

vocfile_header

Next the module's global variables are defined. We'll discuss these variables in more detail later. The vocfile_header variable is initialized with the current character sequence for the CT-Voice format header. In certain situations your program may need to change this sequence. This is why this value hasn't been defined as a constant.

Function prototypes

The next lines contain the function prototypes of the implemented functions. So it's obvious that this module doesn't use the sound sample recording or the user-defined function.

Global error messages

The system of global error messages is handled the same way as in this module's Pascal counterpart.

The global variable used for this purpose is voc_err_stat. When an error occurs with VOCTOOL.H, the error number is assigned to this variable. When the variable contains the value "0", an error isn't present.

You and your program must determine how eventual errors will be handled. In any case, you must ensure that the value of this variable is set back to zero once you've processed the error. Otherwise its value will change only when another error occurs.

The error numbers assigned by VOCTOOL.H are identical to the numbers used in the Pascal version. This is helpful if you want to work with both languages. The following is a list of the individual error numbers:

VOCTOOL error numbers

The 1xx numbers are reserved for errors that occur when the driver file is installed.

Error 100

Error 100 is used when the CT-VOICE.DRV driver file cannot be found.

The directory from which you start a VTTEST program is the directory that will be searched for the driver file. Unlike in the Pascal version, this program won't automatically search in the directory in which the called program is actually located.

To avoid this problem, use the same method used in the Pascal version. Simply determine on which path the called file is located and search for the driver file in that directory. However, you may want to improve the module by including the environment variables.

Refer to Chapter 1 for more information on how to use the environment variables that apply to the Sound Blaster card.

Error 110

Error 110 is used when an insufficient amount of memory is available for loading the driver file into memory.

There are two possible causes. Either all of your PC's available memory is occupied or you're using a memory model that no longer has any memory that can be reserved with the _dos_allocmem() function.

The following program was compiled with the "Small" memory model.

Error 120

Error 120 is released when the file that has been loaded cannot be a CT-VOICE.DRV file. Although the test isn't completely positive, it does compare bytes 3 and 4 of the loaded file with the characters "CT". If the values of these two bytes don't match these characters, you can be sure that this isn't the desired driver file.

As we mentioned in the Pascal unit, you can easily expand this test to obtain more accurate results. The exact character sequence that's used for driver files of Sound Blaster Versions 1.0 through 2.0 and the driver of Sound Blaster Pro is "CT-VOICE Creative Sound Blaster".

2xx error numbers are reserved for errors that may occur when the sound sample files are loaded. These errors are very similar to the ones that may occur while the driver file is being loaded.

Error 200 Error 200 indicates that the specified VOC file couldn't be found. Either you've entered the wrong filename or the file isn't located in the current directory.

Error 210 Error 210 occurs when no more memory is available for loading the sample file. This error can occur for the same reasons as error 110. Since many sample files are very large, some may not fit into your PC's memory even if it's configured optimally.

Error 220 Error 220 is used to indicate that the loaded file doesn't conform to the CT-Voice format. For this identification check, the values of two bytes are checked. The full character sequence common to all formats is "Creative Voice File".

Error 300 Error 300 is reserved for when DOS is unable to free a reserved memory area.

Since C already contains the _dos_freemem() function, a separate function, such as VOCFreeBuffer, didn't have to be written in Pascal. Therefore, error 300 can never occur within the module.

The only reason why it was included is to keep the error numbers between the C and Pascal versions the same. This prevents any confusion and makes it easier to compare the two versions.

If you want error 300 to occur in your C version, you must write a function that sets the error when it detects a failed _dos_freemem() call.

4xx error numbers are used to check whether all hardware parameters are correct once the driver has been loaded successfully. For this, the error procedure uses the results of driver function 3.

Error 400 Error 400 indicates that the driver was unable to find your Sound Blaster card. This function can be used to prevent your program from trying to support the Sound Blaster card on PCs that don't have one.

Error 410 Error 410 is triggered when the port address specified in the driver file, or which was set using driver function 1, doesn't match the card's hardware settings.

Error 420 Error 420 indicates that the interrupt number specified in the driver file, or set through driver function 2, doesn't match the hardware settings of your card.

By using these error messages, it's possible to expand the unit's initialization portion so it actually determines the card's hardware settings automatically.

However, as in the Pascal version, this method doesn't work with several system configurations. So the user should always have the option of specifying these settings independently in case the automatic procedure fails.

5xx error numbers represent errors that can occur during attempts to modify the playback of sound samples.

Error 500

Error 500 indicates that your program has tried to interrupt a playback loop when a loop wasn't being executed.

Error 510

Error 510 occurs when your program tries to pause a sample playback when a playback isn't in progress.

Error 520

Error 520 indicates that your program has tried to continue a paused sample playback, although a sample hasn't been halted with the pause function.

VOCTOOL functions

With the VOCTOOL functions, the individual functions of this module are examined more closely.

_voc_getversion()

The function returns a WORD value. This value contains the main and sub-version number as high and low byte values.

_voc_setport()

You can use this function to change the port address used for the driver initialization before the driver is actually installed. This port address is specified as a WORD value.

Since the driver isn't initialized automatically in the C version, you'll be able to fully use this function even before the driver is initialized for the first time. However, you should permanently record changes to the port and interrupt numbers in the CT-VOICE.DRV file.

_voc_setirq()

This procedure allows you to specify an interrupt number before the driver is initialized. The interrupt number is specified as a WORD value.

The same comments apply to this function as to _voc_setport().

_voc_init_driver()

Unlike in the Pascal version, this function is very important to this module because you must call it at the beginning of your program to load and initialize the driver.

Within the function, the driver file is loaded into memory. Then the file's identity is checked to ensure that it's the correct driver file.

Once this is done, the _voc_getversion() function is used to assign the driver's version number to the global variable voc_driverversion. This allows you to access the driver's version number at any time, without having to call this function again.

Then the driver is initialized. If an error occurs during this process, the global variable voc_err_stat will be assigned the corresponding error number. Only when an error doesn't occur will vocdriver_installed be set to TRUE and the value TRUE be returned by the _voc_init_driver() function.

_voc_deinstall_driver()

This function first switches off the Sound Blaster loudspeaker and then removes the driver from your PC's memory. The memory area previously occupied by the driver will be freed again.

You must always call this function at the end of your program because, unlike the Pascal unit, the C version doesn't execute it automatically.

_print_voc_errmessage()

This function displays the current error as clear text on your screen.

Since only a print() statement without a line feed is used for this output, you'll be able to incorporate this function into your own error message procedure or adapt it to your own needs.

_voc_get_buffer()

_voc_get_buffer() performs all the functions needed for loading a sound sample file into memory.

Simply specify the name of the file that must be loaded. The function then tries to open the file and reserve the necessary

memory, after which the data is loaded into the reserved memory area. The size of this data may be larger than a segment (i.e., 64K).

Once all these tasks have been performed successfully, the function value returns the pointer to the data that has just been loaded. The variable, to which this result must be assigned, must be of the "VOCPTR" or "char far*" type.

If the result of this function is "ZERO", an error has occurred. Then you'll need to evaluate the error number and respond accordingly.

_voc_set_speaker()

With this function you can switch the speaker output of your Sound Blaster card on or off. When "TRUE" is passed as the function parameter, the speaker is switched on and with "FALSE" the speaker is switched off.

When the driver has been initialized successfully, the speaker is switched on automatically. It is switched off automatically when the driver is de-installed.

_voc_output()

This function is used to play back a sound sample currently located in memory. You must specify only the pointer variable identifying the data that should be played back.

Program execution continues immediately after the function has started because the DMA allows the sample playback to occur parallel with other programs. At this point you'll be able to check the global variable voc_status_word within your program.

As long as this variable contains a value that doesn't equal "0", data is still being sent to the DAC (or from the ADC into memory, if you're recording sound samples).

When a marker block is reached during the playback of a sound sample, its contents are copied to the VoiceStatusWord variable. This allows you to check which portion of the sample is currently being played so you can synchronize the playback with other events in your program.

_voc_output_loop()

This function is identical to _voc_output(). However, on some Sound Blaster cards switching on an already active speaker produces crackling sounds. These sounds are particularly annoying when a sound is played back repeatedly in a playback loop.

Because of this, the procedure doesn't switch on the speaker. Ensure that the speaker is switched on when you call _voc_output_loop().

_voc_pause()/_voc_continue()

_voc_pause() is used to pause the playback of a sound sample. Later the playback can be resumed with _voc_continue().

_voc_pause() sets the global variable voc_paused to TRUE, if a sample was being played back at the time the function was called. With this variable, you can check, at any time, whether a playback is currently paused.

The _voc_continue() function then resets voc_paused to FALSE once you continue the playback of the paused sample.

_voc_stop()

This function permanently stops the playback of a sound sample. This is important when you want to start playing a new sample before another sample has been completely played back.

Don't start the playback of a sound sample without first stopping a playback that's in progress. Otherwise, a system crash will occur.

_voc_breakloop()

This function allows you to interrupt an output loop that has been specified by the CT-Voice format within a sound sample file.

You must specify either "0" or "1" as a parameter for this function. This means that the loop will be interrupted either once the current pass has been completed or immediately at the time of the function call. The constants voc_breakend and voc_breaknow have been assigned the corresponding values for this purpose.

Room for improvement

Like the Pascal unit, this module doesn't use two of the driver's functions. These are functions 7 and 13.

The sound sample recording function (function 7) really isn't necessary because other programs can be used for this purpose. Using these programs saves you time. However, if you still need this function, you should be able to use the description from above to integrate the function into the module.

Function 13, the user-defined function, is too specialized to be used at this point. Depending on its purpose and use, you must write a specific function for the given application, which corresponds to the driver's requirements.

Like in Pascal, using the environment variables to locate the driver directories would be an interesting expansion to this C module.

So there are many ways you can expand and improve this version of VOCTOOL.H. For example, you can easily add digital sound to your own game programs.

```
/*
 ********************************************************************
 * Module for controlling the Sound Blaster card in Borland C++ 3.1 *
 *                 using the CT-VOICE.DRV driver.                   *
 ********************************************************************
 *                      (C) 1992 Abacus                             *
 *                    Author : Axel Stolz                           *
 ********************************************************************
*/

#include <io.h>
#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>

#define  FALSE  0
#define  TRUE   1

#define  DWORD  unsigned long
#define  WORD   unsigned int
#define  BYTE   unsigned char
#define  VOCPTR char far*

const char voctool_version[] = "v1.5"; /* Version number of VOCTool      */
const BYTE voc_breakend      = 0;      /* Constant for loop interrupt    */
```

```
const BYTE voc_breaknow      = 1;      /* Constant for loop interrupt    */

   WORD    voc_status_word      = 0; /* Variable for SB status           */
   WORD    voc_err_stat         = 0; /* Variable for driver error number*/
   char    vocfile_header[]     =    /* Variable for CT format header   */
           "Creative Voice File\0x1A\0x1A\0\0x0A\1\0x29\0x11\1";
   BYTE    voc_paused           = 0; /* Flag for voice pause            */
   BYTE    vocdriver_installed  = 0; /* Flag for installed driver       */
   WORD    vocdriver_version    = 0; /* Driver version number           */
   BYTE    vocfile_headerlength = 0; /* Length of CT format header      */
   VOCPTR  voc_ptr_to_driver    = 0; /* Pointer to driver in memory     */

/*
 ******************************************
 * Prototypes for created functions       *
 ******************************************
 */

   void      _print_voc_errmessage(void);
   VOCPTR    _voc_get_buffer(char *voicefile);
   WORD      _voc_getversion(void);
   void      _voc_setport(WORD portnumber);
   void      _voc_setirq(WORD irqnumber);
   WORD      _voc_init_driver(void);
   void      _voc_deinstall_driver(void);
   void      _voc_set_speaker(BYTE OnOff);
   void      _voc_output(char far *bufferaddress);
   void      _voc_output_loop(char far *bufferaddress);
   void      _voc_stop(void);
   void      _voc_pause(void);
   void      _voc_continue(void);
   void      _voc_breakloop(WORD breakmode);

void _print_voc_errmessage(void)
/*
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Displays SB error as text without changing error status.
 */
{
   switch (voc_err_stat) {
     case 100 : printf(" CT-VOICE.DRV driver file not loaded");break;
     case 110 : printf(" No memory free for driver file ");break;
     case 120 : printf(" Wrong driver file ");break;

     case 200 : printf(" VOC file not found ");break;
     case 210 : printf(" No memory free for VOC file ");break;
     case 220 : printf(" File is not in VOC format ");break;

     case 300 : printf(" Memory allocation error occurred ");break;

     case 400 : printf(" No Sound Blaster card found ");break;
     case 410 : printf(" Wrong port address used ");break;
     case 420 : printf(" Wrong interrupt used ");break;
```

```
     case 500 : printf(" No loop in preparation ");break;
     case 510 : printf(" No sample in the output ");break;
     case 520 : printf(" No paused sample available ");break;
      }
}


VOCPTR _voc_get_buffer(char *voicefile)
/*

 * INPUT   : Filename as string
 * OUTPUT  : Pointer to buffer with VOC data
 * PURPOSE : Loads a file into memory and returns the value pointer to
             the data upon being successfully loaded.
 */
{
       int        filehandle;
       long       filesize;
       WORD       blocksize;
       WORD       byte_read ;
       char huge *filepointer;
       VOCPTR     bufferaddress;
       BYTE       check;
       WORD       segment;


/* VOC file not found */
   if (_dos_open(voicefile,O_RDONLY,&filehandle) != 0)
      {
       voc_err_stat = 200;
       return(FALSE);
      }

   filesize = filelength(filehandle);
   blocksize = (filesize + 15L) /16;

/* Insufficient memory for the VOC file */
   check = _dos_allocmem(blocksize,&segment);
   if (check != 0)
      {
       voc_err_stat = 210;
       return(FALSE);
      }

   FP_SEG(bufferaddress) = segment;
   FP_OFF(bufferaddress) = 0;

   filepointer = (char huge*)bufferaddress;

/* Load the sample file */
   do
      {
       _dos_read(filehandle,filepointer,0x8000,&byte_read);
```

```
       filepointer += byte_read ;
      } while (byte_read == 0x8000);

   _dos_close(filehandle);


/* The file is not in VOC format */
   if ((bufferaddress[0] != 'C') || (bufferaddress[1] != 'r'))
      {
       voc_err_stat = 220;
       return(FALSE);
      }

/* Loading successful */
   voc_err_stat = 0;

   vocfile_headerlength = (BYTE)bufferaddress[20];

   return(bufferaddress);
}

WORD _voc_getversion(void)
/*
 * INPUT   : None
 * OUTPUT  : Driver version number
 * PURPOSE : Determines driver version number.
 */
{
   WORD    vdummy;

   asm {
      mov       bx,0
      call      voc_ptr_to_driver
      mov       vdummy, ax
       }
   return(vdummy);
}

void _voc_setport(WORD portnumber)
/*
 * INPUT   : Port address number
 * OUTPUT  : None
 * PURPOSE : Sets the port address before initialization.
 */
{
   asm {
      mov    bx,1
      mov    ax,portnumber
      call   voc_ptr_to_driver
       }
}

void _voc_setirq(WORD irqnumber)
/*
```

```
 * INPUT   : Interrupt number
 * OUTPUT  : None
 * PURPOSE : Sets the interrupt number before initialization.
 */
{
   asm {
      mov    bx,2
      mov    ax,irqnumber
      call   voc_ptr_to_driver
       }
}

WORD _voc_init_driver(void)
/*
 * INPUT   : None
 * OUTPUT  : Error message number, depending on results of initialization
 * PURPOSE : Initializes driver software.
 */

{
       int               filehandle;
       unsigned long     filesize;
       WORD              blocksize;
       char huge        *filepointer;
       WORD              byte_read;
       BYTE              check;
       WORD              segment;
       WORD              vseg;
       WORD              vofs;
       WORD              vout;


   vocdriver_installed = FALSE;

/* Driver file not found */
   if (_dos_open("CT-VOICE.DRV",O_RDONLY,&filehandle) != 0)
      {
       voc_err_stat = 100;
       return(FALSE);
      }
   filesize = filelength(filehandle);
   blocksize = (filesize + 15L) /16;

/* Insufficient memory for driver file */
   check = _dos_allocmem(blocksize,&segment);
   if (check != 0)
      {
       voc_err_stat = 110;
       return(FALSE);
      }

   FP_SEG(voc_ptr_to_driver) = segment;
   FP_OFF(voc_ptr_to_driver) = 0;
```

```
   filepointer = (char huge*)voc_ptr_to_driver;

/* Load the driver file */
   do
     {
      _dos_read(filehandle,filepointer,0x8000,&byte_read);
      filepointer += byte_read ;
     } while (byte_read == 0x8000) ;

   _dos_close(filehandle) ;


/* Driver file doesn't start with "CT", so it's not a CTVoice driver */
   if ((voc_ptr_to_driver[3] != 'C') || (voc_ptr_to_driver[4] != 'T'))
      {
       voc_err_stat = 120;
       return(FALSE);
      }

/* Determine driver version */
   vocdriver_version = _voc_getversion();

/* Start the driver */
   vseg = FP_SEG(&voc_status_word);
   vofs = FP_OFF(&voc_status_word);

   asm {
      mov       bx,3
      call      voc_ptr_to_driver
      mov       vout,ax
      mov       bx,5
      mov       es,vseg
      mov       di,vofs
      call      voc_ptr_to_driver
       }


/* No Sound Blaster card found */
   if (vout == 1)
      {
       voc_err_stat = 400;
       return(FALSE);
      }

/* Wrong port address used */
   if (vout == 2)
      {
       voc_err_stat = 410;
       return(FALSE);
      }

/* Wrong interrupt number used */
   if (vout == 3)
```

```
      {
       voc_err_stat = 420;
       return(FALSE);
      }

   vocdriver_installed = TRUE;
   return(TRUE);
}


void _voc_deinstall_driver(void)
/*
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Switches off driver and releases memory.
 */

{
   if (vocdriver_installed)
      {
       asm {
          mov   bx,9
          call  voc_ptr_to_driver
           }
       _dos_freemem(FP_SEG(voc_ptr_to_driver));
      }
}


void _voc_set_speaker(BYTE on_off)
/*
 * INPUT   : TRUE for speaker on, FALSE for speaker off
 * OUTPUT  : None
 * PURPOSE : Toggles SB card output.
 */
{
   asm {
      mov       bx,4
      mov       al,on_off
      call      voc_ptr_to_driver
       }
}

void _voc_output(VOCPTR bufferaddress)
/*
 * INPUT   : Pointer to sample data as pointer
 * OUTPUT  : None
 * PURPOSE : Plays back a sample.
 */
{
   WORD  vseg;
   WORD  vofs;

   _voc_set_speaker(TRUE);
```

```
   vseg = FP_SEG(bufferaddress);
   vofs = FP_OFF(bufferaddress) + vocfile_headerlength;

   asm {
      mov       bx,6
      mov       es,vseg
      mov       di,vofs
      call      voc_ptr_to_driver
       }
}


void _voc_output_loop(VOCPTR bufferaddress)
/*
 *    Difference to _voc_output :
 *    Speaker is not switched on before output of each sample, since this
 *    can lead to crackling noise on some Sound Blaster cards when playing
 *    back a loop.
 */

{
   WORD  vseg;
   WORD  vofs;

   vseg = FP_SEG(bufferaddress);
   vofs = FP_OFF(bufferaddress) + vocfile_headerlength;

   asm {
      mov       bx,6
      mov       es,vseg
      mov       di,vofs
      call      voc_ptr_to_driver
       }
}

void _voc_stop(void)
/*
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Stops a sample.
 */
{
   asm {
      mov       bx,8
      call      voc_ptr_to_driver
       }
}

void _voc_pause(void)
/*
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Pauses a sample.
```

```
 */
{
   WORD  pcheck;

   voc_paused = TRUE;

   asm {
      mov       bx,10
      call      voc_ptr_to_driver
      mov       pcheck,ax
       }
   if (pcheck == 1)
      {
       voc_paused = FALSE;
       voc_err_stat = 510;
      }
}

void _voc_continue(void)
/*
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Continues a paused sample.
 */
{
   WORD  pcheck;

   asm {
      mov       bx,11
      call      voc_ptr_to_driver
      mov       pcheck,ax
       }
   if (pcheck == 1)
      {
       voc_paused = FALSE;
       voc_err_stat = 520;
      }
}

void _voc_breakloop(WORD breakmode)
/*
 * INPUT   : Break mode
 * OUTPUT  : None
 * PURPOSE : Interrupts a sample loop.
 */
{
   asm {
      mov       bx,12
      mov       ax,breakmode
      call      voc_ptr_to_driver
      mov       breakmode,ax
       }
   if (breakmode == 1) voc_err_stat = 500;
}
```

VTTEST

VTTEST.C is almost identical in function to its Pascal counterpart. However, if you've compiled your program using the Small memory model, you won't need to specify an upper memory limit within your program, unlike in the Pascal version.

So, the program begins by linking the CONIO.H file, in which all the functions that will later be needed to read the keyboard input are defined. Then the VOCTOOL.H header file is added to the program.

Running VTTEST.EXE

Make sure that the CT-VOICE.DRV driver is either in the same directory as VTTEST.EXE, or accessible. Make sure the DEMO.VOC file is in the same directory as VTTEST.EXE (copy DEMO.VOC from the \ABACUS\SBBOOK\TP60\VOCTOOL directory). Run VTTEST.EXE.

textnumerror function

The textnumerror function is called when an error is detected within the test program. This function displays the error's number and corresponding text on the screen.

Then the program is interrupted using the exit function and the detected error number is passed to DOS as Errorlevel.

This allows you to check, for example from within a batch file, whether the program has been interrupted because of an error or whether it was executed without any errors. If no errors occurred, the Errorlevel is zero.

After the screen is cleared, _voc_init_driver() is called by the main program. Although this wasn't necessary in the Pascal version, don't forget this step in your C programs. The initialization has been performed successfully only when this function returns the value TRUE. Then the vocdriver_installed global variable is also set to TRUE.

If the initialization has failed, the program is interrupted with the textnumerror() function. This allows you to determine, for example, whether your own programs will be allowed to operate on a system that doesn't have a Sound Blaster card.

In the program, the variable voc_buffer should point to the sound sample data located in memory. So the value returned by _voc_get_buffer(DEMO.VOC) is assigned to this variable. If this value is NULL, then an error has occurred while the file was being

loaded. The program is then interrupted together with the corresponding error message.

Once the driver and the file have been loaded successfully, the driver's version number is displayed on the screen and the user is asked to enter either S or M, depending on whether the sound must be played single or multiple times.

The switch() statement then branches to the program portion specified by the user's choice.

Single playback

In the "s" portion, first the help text is displayed, informing the user that the playback can be stopped by pressing any key. Then the playback is started with the _voc_output() function.

Since the procedure continues immediately after the playback has been started, the main program is sent into a do... while wait loop. This loop checks whether a key has been pressed to stop the playback or whether the end of the sample has been reached.

To determine this, the contents of the variable voc_status_word are checked. If keyboard input is detected, the playback of the sample is stopped.

Multiple playbacks

The "m" portion repeatedly plays the sample until the user presses Esc. This is done by using _voc_output_loop() to play back the sample.

Ensure that the Sound Blaster speaker is switched on whenever you use the _voc_output_loop() function. In our program, the speaker is switched on when the driver is initialized.

If you've switched off the speaker at any point within your program, you must switch it back on before calling _voc_output_loop().

The do... while wait loop in this section again checks whether a key has been pressed or whether the end of the sample has been reached.

If a key has been pressed, its ASCII value is assigned to the variable ch. If this value doesn't contain the value 27 (= ASCII code for Esc), the outer do... while loop is started again and the sample is played back again from the beginning.

As we mentioned for the Pascal version, these playback loops are useful for adding music to graphics.

Using the Voice Editor, you'll be able to edit short digitized segments of music so it's impossible to determine when the sample ends and when it begins again.

By using these methods, you can easily create a program that's similar to the Arranger program presented in the Pascal section.

Ending the program

If the playback has been interrupted by the user or the program has completed the single playback, the memory area identified by the voc_buffer pointer is freed. This is done with the _dos_freemem() function.

Then _voc_deinstall_driver() is called. Unlike the Pascal version, this is necessary because in this module the driver isn't deactivated automatically and the driver memory area isn't freed automatically.

Once all the necessary tasks have been performed, the program can end.

With this small program you can start to write your own sound programs in C. You'll find many ways to use the information we've just presented.

```
/*
 **********************************************************************
 *   Demonstration program  for VOCTOOL unit, (W) in Borland C++ 3.1     *
 **********************************************************************
 *                          (C) 1992 Abacus                              *
 *                         Author : Axel Stolz                           *
 **********************************************************************
 */

#include <conio.h>
#include "voctool.h"

void textnumerror()
/*
 * INPUT   : None, the data come from the voc_err_stat global variable
 * OUTPUT  : None
 * PURPOSE : Displays SB error as text on the screen. Includes the error
             number and ends the program with the error level that
             corresponds to the error number.
*/

 {
   printf(" Error # %d: ",voc_err_stat);
   _print_voc_errmessage();
   exit(voc_err_stat);
 }
```

```
void main()
{
  VOCPTR voc_buffer = 0;
  char   ch;

  clrscr();

  if (_voc_init_driver() == FALSE) textnumerror();

/* Loads the DEMO.VOC file into memory */
  voc_buffer = _voc_get_buffer("demo.voc");

/* VOC file could not be loaded */
  if (voc_buffer == NULL) textnumerror();

/* Main program loop */
  printf("CT-Voice Driver Version : ");
  printf("%d.%02d\n",vocdriver_version >> 8, vocdriver_version % 256);
  printf("(S)ingle play or (M)ultiple play?\n");
  printf("Press a key: ");
  ch = getche();
  printf("\n\n");
  switch (ch) {
     case 's': printf("Press a key to stop the sound...");
               _voc_output(voc_buffer);
               do {} while ((voc_status_word != 0) && (kbhit() == 0));
               if (kbhit() != 0) _voc_stop();
               break;
     case 'm': printf("Press <ESC> to cancel...");
               ch = ' ';
               do {
                   do {} while ((voc_status_word != 0) && (kbhit() == 0));
                   if (kbhit() != 0) ch = getch();
                  } while (ch != (char)27);
               _voc_stop();
               break;
     }
/* Free VOC file memory */
  _dos_freemem(FP_SEG(voc_buffer));

  _voc_deinstall_driver();
}
```

5.2.5 Sound programming under Windows

Sound Blaster and Windows

Windows offered only limited support of sound cards in Version 3.0. However, with the release of Multimedia Extensions and Windows Version 3.1, Windows now provides full support of sound cards.

WAV standard

Windows uses the WAV file format for sound sample files. In this section we'll discuss how to play back sound files of this format.

As you'll see, the functions provided by Windows are very powerful. Because of this, you shouldn't have any problems when creating WAV files using Turbo Pascal for Windows, Borland C++, and Visual Basic.

WAV format has a different structure from CT-Voice format. Since it's intended to have the same basic function on numerous sound media, this format isn't tailored specifically to the Sound Blaster system. However, it does have the advantage of being a Windows standard. This means that all WAV files can be played on all sound cards that support Windows.

RIFF format

Microsoft has created a basic file structure for most of the file types used with Windows. This makes it easier to use different formats simultaneously.

If you've worked with the DPaint application, you've already used files that conform to this structure.

Electronic Arts has developed a system called IFF, which is also used by DPaint. IFF is an abbreviation for "Interchange File Format". The Commodore Amiga uses IFF graphic files, IFF text files, and IFF sound sample files. However, PCs usually use only the DPaint IFF graphic files, which are identified by the ".LBM" filename extension.

The advantage of this format is that DPaint can be used to read Amiga graphic files without converting them and vice versa.

RIFF and chunks

Microsoft used this concept as the basis for its RIFF (Resource Interchange File Format). A RIFF file consists of several smaller file components called *chunks*.

A chunk is a logical data segment that always has the same structure, regardless of what type of data it contains.

RIFF rules

It's even possible to add your own chunks, if you want to expand the format. However, to do this, you must follow certain rules.

To retain the original format specified by Microsoft, we'll present all the examples in this section in C. However, all data types can easily be used in Pascal.

The structure of a chunk looks like this:

```
typedef unsigned long DWORD;
typedef unsigned char BYTE;
typedef DWORD FOURCC;          // Type the chunk name

typedef struct {
   FOURCC  ckID;
   DWORD   ckSize;             // Chunk size
   BYTE    ckData[ckSize];     // Chunk data array
} CK;
```

FOURCC

The type definition FOURCC (Four Character Code) indicates that a chunk always starts with a sequence of four ASCII characters, which are used to identify the chunk.

So the name of a chunk must contain four characters. If you want to use fewer characters, you must fill the remaining characters to the right with spaces (ASCII 32). However, spaces aren't allowed within the name (i.e., between characters). Also, the name must begin with a letter.

A chunk could be named SBP1 or SBP2. SBP_ is allowed, but names such as SB_P or _SBP aren't allowed.

When working with the chunk name, you may also want to interpret the four ASCII characters as a 32-bit number and use this value to perform a name comparison.

Within the defined structure, CK, the chunk's name (CK) is referred to as the structure element ckID.

The element chSize contains the size of the chunk as a positive 32-bit number. For this purpose, the type DWORD (Double Word) has been defined.

However, the size contained in DWORD doesn't include the four bytes of ckID and ckSize. So this value represents only the number of bytes starting after chSize. The chunk's data are then placed in ckData[].

Remember that the number of bytes contained in ckData[] must always be accessible in words. This means that when an odd number of data bytes is present, an extra pad byte is added to the end of ckData[]. This byte contains the value zero and isn't included in ckSize.

Advantages of the IFF system

One advantage of the chunk system that's used by the RIFF format is that programs can simply skip unrecognizable chunks and

continue their work with the next chunks. This is possible only through the structure previously shown.

However, be sure that when a chunk containing an odd number of bytes is read, the pad byte is also read. Otherwise you'll never find the beginning of the next chunk.

By using this method, you can write programs that are capable of reading only the recognizable chunks contained in RIFF files and can then further process this data.

Format restrictions

However, Microsoft has some restrictions on creating and naming new chunks. To prevent confusion, new chunks consisting of uppercase letters and numbers must be registered with Microsoft.

Microsoft made this decision to ensure consistency among programmers.

So, chunk names consisting of only uppercase letters and numbers, such as the previous examples, must be registered. However, names such as sbp1, sbp2, or sbp_ don't have to be registered. Chunks using lowercase letters can be used as special chunk types for individual use. So lowercase letters indicate that this chunk type might have a different function and meaning in other situations.

Only the chunk types LIST and RIFF can contain other chunks within their data section. Since the other chunks don't have additional chunks within their data segments, these two chunks have a slightly different structure. They have another FOURCC name, which describes the type of RIFF file, after their data length specification. For example, this could be the character sequence WAVE.

There are five types of RIFF files:

RIFF Name	File type
PAL	Palette file
RDIB	Device-independent bitmap
RMID	MIDI format file
RMMP	Multimedia Movie file
WAVE	Sound sample file

For detailed information about the structure and definition of existing RIFF elements, refer to the *Microsoft Windows Software Development Kit Multimedia Programmer's Reference*.

The WAVE format

Microsoft has developed its own format for describing RIFF file structures. This format is easy to understand if you know what the format is and how it's used.

The following is Microsoft's description of WAVE file format:

```
<WAVE-form>    -> RIFF(   'WAVE'              // Form type
                          <fmt-ck>            // Waveform format
                          <data-ck>           // Waveform data

<fmt-ck>       -> fmt(    <wave-format>       // WaveFormat structure
                          <format-specific> ) // Dependent on format
                                              //    category

<wave-format> -> struct {
                  WORD    formatTag;          // Format category
                  WORD    nChannels;          // Number of channels
                  DWORD   nSamplesPerSec;     // Sampling rate
                  DWORD   nAvgBytesPerSec;    // For buffering
                  WORD    nBlockAlign;        // Block alignments
                    }

<PCM-format-specific> ->
                    struct {
                        UINT   nBitsPerSample;      // Sample size
                                }

<data-ck>      -> data <wave-data> )
```

This means that the WAV file begins with the keyword RIFF. This is followed by the 32-bit length specification, just like in the structure of a chunk. After this you'll see the text WAVE.

<fmt-ck> The chunk, which is referred to as <fmt-ck> in the above description, begins immediately after this. This chunk is called fmt_.

<wave-format> The chunk's data block starts after the 32-bit length value. The <wave-format> data structure is contained in this data block.

In the Microsoft description, this structure has been arranged according to the C convention. In this configuration, its length is 14 bytes.

formatTag

The first element, formatTag, is important to the interpretation of the sample data located in the <data-ck> chunk.

PCM format

Currently the PCM format is being used for WAV files. PCM, which is an abbreviation for Pulse Code Modulation, refers to a specific encoding format for sound samples.

The value 1 in formatTag indicates that the file contains PCM data. In Windows, the constant WAVE_FORMAT_PCM is used for this value.

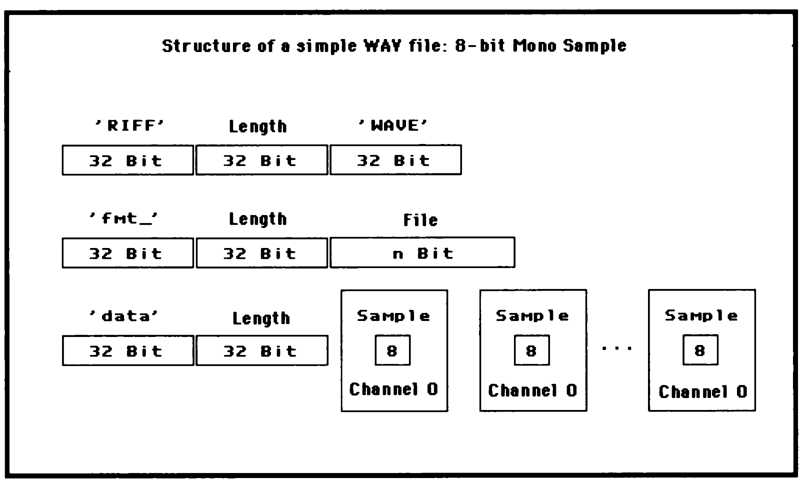
<format-specific>

The data structure <format-specific> is included in the fmt_ chunk to describe this format. The data structure specifies how many bits of the data block represent a single sample or scanning value.

The Sound Blaster card uses 8 bits per sample. However, 12- and 16-bit samplers are also available. The WAVE format supports only 8- and 16-bit samples. This means that for samplers using 12 bits, the WAVE format will still record 16 bits per sample, to avoid unnecessary calculations.

8-bit mono

In an 8-bit mono sample, all bytes are stored consecutively. Channel 0 is used for mono samples.

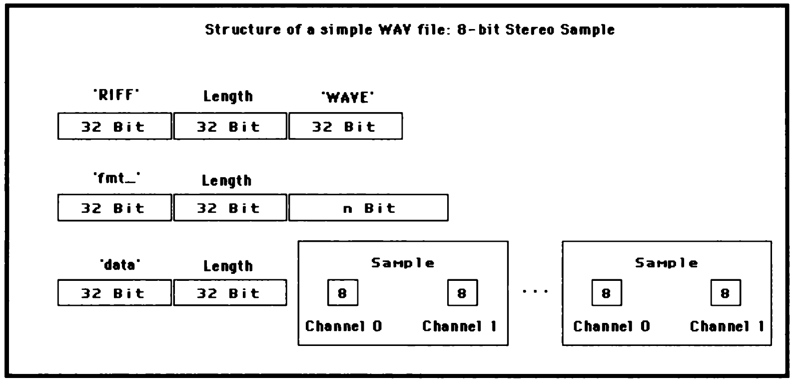


Structure of a simple 8-bit mono file

8-bit stereo

In a stereo sample, channel 0 is the left channel and channel 1 is the right. In an 8-bit stereo sample, data is stored according to the format: 8 bits for channel 0, 8 bits for channel 1, 8 bits for channel 0, 8 bits for channel 1..., etc.

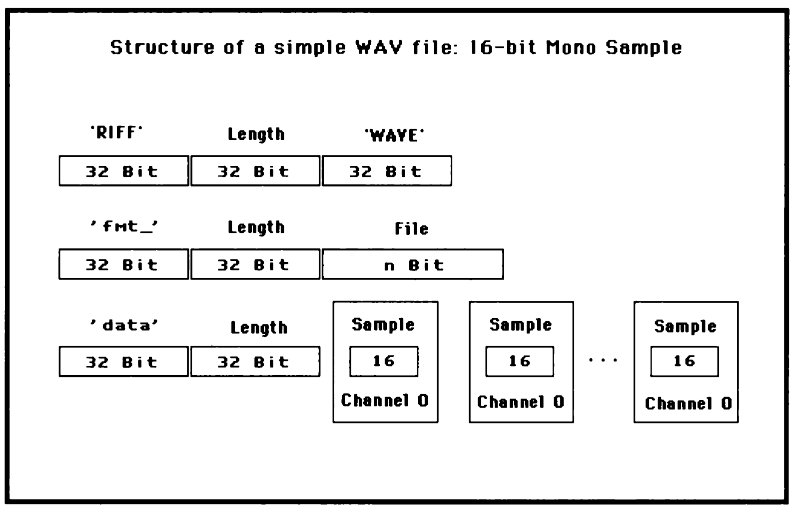
This means that a single sample (scanning value) consists of two bytes, one for the left channel and one for the right, always in alternating sequence.



Structure or a simple 8-bit stereo file

16-bit mono

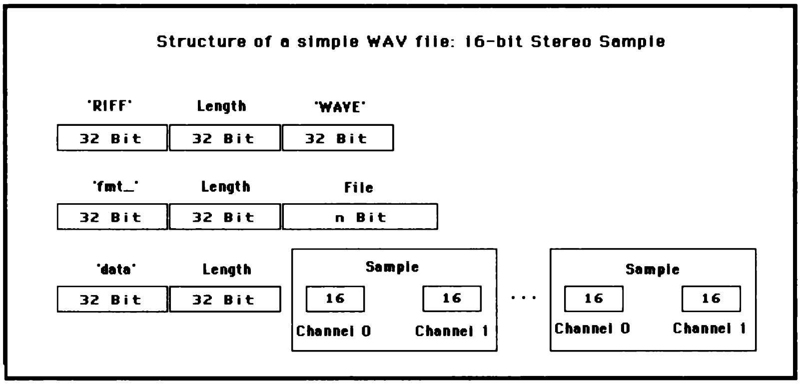
To represent a 16-bit mono sample in memory, 2 bytes are needed to record a single sample. Otherwise, the sequence of bytes is identical to the 8-bit mono sample.



Structure of a simple 16-bit mono file

16-bit stereo

A 16-bit stereo sample requires the most memory for each sample. Exactly 4 bytes are needed (two for the left channel and two for the right). The sequence of bytes is always one low byte and one high byte for each channel.



Structure of a simple 16-bit stereo file

In an 8-bit sample, the values of each byte lie between 0 and 255. So a value of 128 represents the median of the digitized wave form.

For a 16-bit sample, these values range from -32768 to +32767. In this case, the value 0 forms the median.

With the information on the minimum and maximum byte values for each sample you'll be able to display the wave form in a WAV file graphically on your screen. To do this, simply create a curve from these values.

nChannels

Now let's return to the <wave-format> structure. The value following formatTag is nChannels.

This value indicates the number of channels for which the data in the file has been encoded. If this value is 1, the Byte sequence in <data-ck> has been recorded in mono; if it is 2, the data are in stereo.

nSamples PerSec

The element nSamplesPerSec specifies the file's sampling rate. Windows supports sampling rates of 11025 Hz, 22050 Hz, and 44100 Hz.

nAvgBytes PerSec

The value in nAvgBytesPerSec (average bytes per second) can be very useful for playing back samples. A program can use this value to determine the size of the buffer that will be needed to output the sample.

nBlock Align

The value contained in nBlockAlign indicates how many bytes are used to represent a single sample in that particular WAV file.

For an 8-bit mono sample, the block alignment value is 1, and for a 16-bit stereo sample it's 4, as we previously mentioned in the description of the byte sequence for the different sample formats.

<format-specific>

The data structure <format-specific> is next. We explained this structure in conjunction with formatTag.

Although this structure is available only for PCM format, eventually it may be expanded for other formats. For PCM format, its length is 2 bytes.

This completes the fmt_ chunk. Its length is currently 16 bytes; 14 bytes belong to the <wave-format> data structure and 2 bytes belong to the <format-specific>. However, the chunk's actual length can always be determined through the 32-bit length value.

<data-ck>

Next is the data chunk, which contains the individual sample data.

The information from chunk fmt_ are needed for decoding the information contained in this chunk. Therefore, every WAV file must always contain at least both of these chunks. However, other chunks can also be added to the file.

So any program that must process the WAVE format must be able to recognize and decode at least the fmt_ and data chunks. All other chunks in the file can, at will, be ignored and skipped.

A WAVE example

The following is an example for the description of a WAV file, according to the Microsoft description standard:

```
RIFF('WAVE' fmt (1, 2, 11025, 22050, 2, 8)
            data( <wave data> ) )"
```

These lines represent an 8-bit stereo file with a sampling rate of 11,025 Hz.

If this information is too technical for you, don't worry because you'll still be able to use WAV files.

You must thoroughly understand this format only if you want to write programs, under Windows, which use WAVE files and which read and play the samples located in these WAVE files.

5.2.6 Sample playback in Turbo Pascal for Windows

It's easy to play back WAV samples because Windows performs almost all the programming steps required for this task.

MMSYSTEM

The Dynamic Link Library MMSYSTEM.DLL contains an entire row of functions that enable you to play back sound samples under Windows.

For WAV file playback, Windows provides high-level audio services and low-level audio services. High-level audio services are functions that perform almost all the tasks for you. With the low-level audio services, you must perform more of the required tasks. However, using these functions gives you more freedom and a better overview of the process.

The playback of any desired WAV file actually requires only a single high-level audio services function, called SndPlaySound.

The name of the WAV file or a pointer to a memory area, together with a value indicating how the data must be interpreted, are simply passed to the function; nothing else is required.

You need only the MMSYSTEM.PAS unit, as the following shows, to access the functions contained in MMSYSTEM.DLL.

```
UNIT MMSystem;
{
 ***************************************************************************
 * Microsoft Windows 3.0 + MM Extensions / Windows 3.1 Sound Interface Unit*
 ***************************************************************************
 *            Copyright (C) 1992 Abacus                                    *
 *                             Authors :                                   *
 *                       Thorsten Petrowski                                *
 *                            Axel Stolz                                   *
 ***************************************************************************
 *         WAV file output with Turbo Pascal for Windows 1.0              *
 ***************************************************************************
}

INTERFACE

CONST
   SND_SYNC      = $0000;   { Play synchronous sound (default)      }
   SND_ASYNC     = $0001;   { Play asynchronous sound               }
   SND_NODEFAULT = $0002;   { If no WAV files exist,                }
                            { do not use INI sounds                 }
   SND_MEMORY    = $0004;   { Pointer points to data in memory      }
   SND_LOOP      = $0008;   { Repeats sound until next SndPlaySound }
   SND_NOSTOP    = $0010;   { Do not interrupt playing sound        }
```

```
PROCEDURE SndPlaySoundFile (Name: PChar; Flags: WORD);
{
 * INPUT   : Name as pointer to string ended with NULL, Flags as WORD
 * OUTPUT  : None
 * PURPOSE : Sends a WAV file using the MMSystem.DLL library.
}

PROCEDURE SndPlaySoundMem (MemP: Pointer; Flags: WORD);
{
 * INPUT   : MemP as pointer to memory range, flags as WORD
 * OUTPUT  : None
 * PURPOSE : Sends a WAV file to memory using the MMSystem.DLL library.
}

IMPLEMENTATION

PROCEDURE SndPlaySoundFile (Name: PChar; Flags: WORD);
                            EXTERNAL 'MMSYSTEM' INDEX $0002;
PROCEDURE SndPlaySoundMem  (MemP: Pointer; Flags: WORD);
                            EXTERNAL 'MMSYSTEM' INDEX $0002;

BEGIN
END.
```

As the listing shows, it's only necessary to make the SndPlaySound procedure available within the unit's interface section and to insert the reference to the DLL in the implementation section. This is accomplished through the supplement EXTERNAL 'MMSYSTEM' INDEX $0002.

This informs the compiler that it must search for the specified procedure in MMSYSTEM.DLL and that the command index number is $0002.

MMSystem DLL structure

Each command within MMSYSTEM.DLL has its own index number. To determine which index number belongs to the desired command, you can use a hex editor, for example, to scroll through the MMSYSTEM.DLL file and search for the text "SndPlaySound".

However, it's also possible to write a short Pascal program that searches the file for a keyword and then returns the corresponding index number.

You can also use this procedure to access the other functions of any DLL. If you do this, ensure that the type and number of parameters required by a DLL function correspond with the particular function implementation.

If necessary, it would also be possible to implement other MMSystem functions in the MMSYSTEM.PAS unit. For example,

you can incorporate MCI (Media Control Interface) functions. These functions are also capable of playing back WAV sound samples.

Unfortunately, we cannot list and explain other MMSystem functions and their parameter structures because this is beyond the scope of this book. The functions contained in MMSYSTEM.DLL present enough information to fill an entire separate book. However, in most cases the SndPlaySound function is sufficient.

Therefore, the framework of this Windows example has been limited to the exploration of the SndPlaySound function. You can use this function to play back any WAV sound sample that will fit into your PC's memory. This should easily be possible with sound samples of up to 100K.

If you need more information, refer to the Microsoft Multimedia Extensions documentation. This provides detailed descriptions of all the data types, functions, and procedures needed for Multimedia programming.

SndPlaySound gives you access to an entire range of sound playback applications under Windows.

Implementation

To simplify using SndPlaySound, we've used the function in two different ways in this unit.

The first parameter required by SndPlaySound is a zero-terminated string. SndPlaySoundMem requires a pointer to the memory area containing the sound sample data.

Actually the function is only named SndPlaySound. However, by specifying the index number, the names SndPlaySoundFile and SndPlaySoundMem are also assigned correctly.

We'll use the specifications File and Mem only when the command's use depends on the difference between these two implementations.

After the SndPlaySoundFile call, the function first searches the [Sounds] section of the WIN.INI file. If the specified sound isn't found there, the function will try to locate a corresponding file.

If this file cannot be found, the SystemDefault sound contained in WIN.INI is played back.

Different sound parameters

The parameters you specify determine how the sound samples in WAV files are played back. The flags that modify the sample playback are described.

The parameters are additive. This means that you can add further parameters to flags, such as SND_SYNC or SND_ASYNC, using the OR operator and then pass the resulting parameter to SndPlaySound.

To simplify using these parameters, the flags required for modifying the playback of WAV files have been defined as constants in the MMSYSTEM.PAS unit.

SND_SYNC parameter flag

This parameter is basically the standard value and has the value 0. When this flag is set, the execution of the remaining program is halted until the playback of the WAV file has been completed. Since the sound playback occurs synchronously with the program execution, this flag has been abbreviated as SYNC.

SND_ASYNC parameter flag

This parameter is the exact opposite of SND_SYNC and has the value 1. When this flag is set, program execution will continue immediately after the playback of the WAV file has been started. The program and sound playback are therefore processed simultaneously or asynchronously.

SND_NODEFAULT parameter flag

When SndPlaySound is called with a filename that Windows cannot find, a system tone contained in WIN.INI, under the heading "[sounds]", is automatically emitted. If you add the flag SND_NODEFAULT to one of the two parameters (SND_SYNC or SND_ASYNC), this sound won't be emitted if the desired sound file cannot be found.

SND_MEMORY parameter flag

You can set this flag when you want to specify a pointer, as a function parameter, to a memory area containing a loaded WAV file, instead of to a character string with a filename.

You must set this flag when using SndPlaySoundMem and when the pointer is pointing to a WAV data memory area. Otherwise,

the procedure is the same as the function call while using a filename.

SND_LOOP parameter flag

This flag specifies that the sample must be played back repeatedly until the entire playback of samples is stopped or until another sample is started.

SND_LOOP should be used only in conjunction with the SND_ASYNC flag, not SND_SYNC.

SND_NOSTOP parameter flag

When this flag is set, the specified sound sample is started only if no other file is currently being played back. So this flag prevents you from interrupting another sample that has been started with the SND_LOOP flag.

We'll present some examples to help you understand how to use these flags.

Start the asynchronous output of the WAV file ALARM.WAV by using the line:

```
SndPlaySoundFile('\ABACUS\SBBOOK\WAV\ALARM.WAV',SND_ASYNC OR SND_LOOP);
```

This file is played back continuously in a loop. So the alarm is played back repeatedly until another file is started or until a call is made to:

```
SndPlaySoundMem(NIL,SND_SYNC);
```

The sound is repeated until it's stopped, even if your program has already ended.

You could also use a command line like:

```
SndPlaySoundFile('\ABACUS\SBBOOK\WAV\SEALION.WAV',SND_ASYNC OR
                 SND_LOOP OR SND_NODEFAULT OR SND_NOSTOP);
```

This function call starts the playback of SEALION.WAV, if another playback isn't currently in progress. The playback is looped continuously, and if the file isn't located the default sound isn't emitted.

Ideas for sound in your programs

The Turbo Pascal program for Windows listed later illustrates how you can use sound, for example, to enhance the click of a command button on your screen.

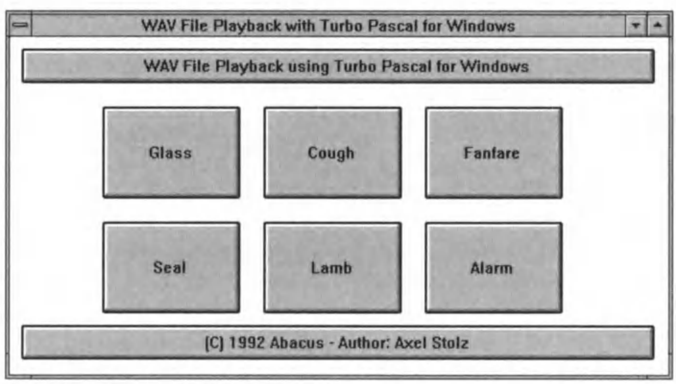
In this case, the only limitation is your imagination. If you want, you could add a specific sound to the selection of each menu item.

For example, your Windows environment could greet you with the calm words "I'm completely operational and all my circuits are functioning perfectly." (If you've viewed the film *2001: A Space Odyssey*, these words probably sound familiar.) Or when you select the *About...* menu item, a drumroll, a cymbal crash, and a fanfare could introduce the About dialog box.

You may want to spend some time experimenting to determine how you want to use sound in your own programs. Also, you should include an option for switching off the program sound for users who don't want to use sound.

TPWSOUND

The TPWSOUND.EXE Windows application creates a window containing buttons; each button produces a different sound. Six of the buttons are labeled with the names of their corresponding sounds and two buttons are used, as a top and bottom bar, for extra audio/visual effect.



TPWSOUND window

The program was written as object-oriented, using the Windows objects provided by Turbo Pascal for Windows. If you've worked with Turbo Pascal for Windows before, you'll be able to recreate this window easily.

Only the procedures of the TSoundWindow that begin with "Sound" are important. These procedures contain the actions that are executed when one of the buttons is clicked.

In this example, the WAV files are searched for under the \ABACUS\SBBOOK\WAV\ path. If you've installed the programs on another directory or another driver, you must modify the program accordingly.

Five of the eight buttons simply use the SndPlaySoundFile function to load a WAV file into memory and play back this file. One exception is the SoundAlarm procedure because this sound plays in a loop. Therefore, the parameter SND_ASYNC or SND_LOOP is used with this function call.

The three other buttons use a different method of playing back their respective sound samples.

Sounds from WIN.INI

For example, when you click the title bar, the SoundSysStart procedure is called. This procedure starts a sound sample called SystemStart, which is located in WIN.INI.

This sample is usually played when you start Windows. You determine which sound is actually represented by this sample name. This is done by opening the Windows Control Panel and selecting the Sound icon from within the Control Panel window.

Here you can assign a WAV sound to different Windows events. These changes are then recorded in the WIN.INI file. It's also possible to use your Windows editor to modify WIN.INI directly.

Sounds from memory

When the [Lamb] button is clicked, the SoundLamb procedure is called. During the TSoundWindow initialization, this WAV file is already loaded into memory. So this file doesn't have to be loaded by the SoundLamb procedure.

Therefore, for the SndPlaySound procedure you must specify only at which location in memory this sample is located. The SndPlaySoundMem function is used to do this.

The pointer to the LambArray array is passed to the function instead of the Filename character pointer and the SND_ASYNC parameter flag is combined with the SND_MEMORY value.

Loading WAV files into an array is one way of storing WAV data in memory. However, this isn't the best method for Windows.

Instead, it's better to link WAV data that will be used in a program to that program by using a resource.

Sounds from a resource

First you must create a resource that can contain the WAV data. However, the Whitewater Resource Toolkit doesn't support the use of WAVE resources. So you must create these resources manually.

To do this, you need a resource script file, which you can create by simply using an ASCII editor.

The script file used in this demo program is named TPWSOUND.RC. It contains only one entry, since only one WAV file is used in this example:

```
Fanfare WAVE \abacus\sbbook\wav\fanfare.wav
```

This specifies that the Fanfare icon will be assigned a resource of the type WAVE and that the required data are located in the file FANFARE.WAV, which is located in \ABACUS\SBBOOK\WAV\. Then you must use RC.EXE to compile this script file.

Resource compiler

RC is the resource compiler. It's a part of Turbo Pascal for Windows and is located in the \TPW\UTILS\ directory.

The resource compiler creates the TPWSOUND.RES file, which now contains the WAV data. By using the following compiler directive, you can link this resource to your program:

```
{$R TPWSOUND.RES}
```

Step 1: FindResource

The SoundFanfare procedure illustrates how this data is accessed from within the program.

You'll need a pointer variable, which will then point to the resource memory (WavResPtr), as well as two handle variables, which are needed to allocate memory to this resource (WavHandle and WavResInfo). First you must search the memory for the resource by using the FindResource function.

First the variable HInstance is passed to this function. This variable contains the handle of the instance of the demo program, if it's been loaded into the Windows environment.

The second parameter consists of a string containing the name that's been assigned to the WAV data in the resource script file. In our demo program, the name is Fanfare.

The third parameter is the resource's type, which is also specified within the resource script file. This is the type WAVE.

The variable WaveResInfo contains a numeric value after the function call. If this value is zero, the specified resource couldn't be found.

Step 2: LoadResource

Next the function LoadResource is needed. The variable HInstance is again passed as a function parameter and the just returned handle WavResInfo is used as the second parameter. The function loads a resource into memory, if necessary, and returns a handle to this resource as its function value.

If the resource was already located in memory when LoadResource was called, only the handle is returned. In the demo program this handle is assigned to the variable WavHandle. If WavHandle contains zero following the function call, then the specified resource doesn't exist.

Step 3: LockResource

Windows rearranges resources in memory, depending on how full your PC's memory is at any given time. However, this shouldn't occur during the playback of a WAVE resource.

If this does occur, the most harmless result would be that Windows would abort the program due to a general protection violation. However, the loudspeaker of your sound card would now emit only audio garbage.

To prevent this from happening, the resource must be held at its current location in memory. This is done with the LockResource function.

The handle just assigned to WavHandle is used as the parameter for this function. The result of the function is a pointer to the memory area currently containing the sound data. This pointer is assigned to WavResPtr.

Once this pointer exists, the playback of the WAV data can begin. The demo program uses the SndPlaySoundMem procedure for this purpose with its SND_ASYNC and SND_MEMORY flags. This allows the sound to be played back from the memory area allocated to the resource.

Use UnlockResource carefully

It's important to activate the program line containing UnlockResource only when using the SND_SYNC sound flag. If you don't do this, the resource data may be "stolen" from memory during the sample playback. This occurs for basically the same reasons for which LockResource was executed.

If you're using SND_SYNC, your program is paused until the playback has ended. After this point you'll be able to free the resource with UnlockResource.

After experimenting with the program, you may want to swap a few sounds for other ones, or add more buttons or an entire menu. For example, you could add a file dialog box, in which the user could select the sound files that must be assigned to the individual buttons.

However, you may also want to begin applying this information to your own programs.

All the items we discussed above are included in the following listing of TPWSOUND.PAS.

```
Program WinSound;
{
 ***************************************************************************
 *   Demonstration application for MMSystem unit, Turbo Pascal for Win 1.0 *
 ***************************************************************************
 *                     Copyright (C) 1992 Abacus                           *
 *                        Author : Axel Stolz                              *
 ***************************************************************************
 * The SndPlaySound statement provides easy play for WAV files.            *
 ***************************************************************************
}

{$R TPWSound.Res} { Include resource containing 'Fanfare' sound name }

USES
   WinTypes, WinProcs, WObjects,  Strings, MMSystem;

{ * Get constants for different buttons }
CONST
   ID_SysStart= 100;
   ID_Glass   = 101;
   ID_Cough   = 102;
   ID_Fanfare = 103;
   ID_Sealion = 104;
   ID_Lamb    = 105;
   ID_Alarm   = 106;
   ID_Sting   = 107;
```

```
TYPE
   TSoundApp = OBJECT(TApplication)
      PROCEDURE InitMainWindow;virtual;
      END;

   PSoundWindow = ^TSoundWindow;
   TSoundWindow = OBJECT(TWindow)
      LambArray : ARRAY[0..7000] OF CHAR;  { Buffer memory for lamb sample }
      BSysStart,                           { Variables for eight buttons   }
      BGlass,
      BCough,
      BFanfare,
      BSealion,
      BLamb,
      BAlarm,
      BSting    : PButton;
      CONSTRUCTOR Init (AParent : PWindowsObject; ATitle : PChar);
      { Assign ID constants for button procedures }
      PROCEDURE SoundSysStart (VAR Msg : TMessage);
                              virtual ID_First+ID_SysStart;
      PROCEDURE SoundLamb     (VAR Msg : TMessage);
                              virtual ID_First+ID_Lamb;
      PROCEDURE SoundFanfare  (VAR Msg : TMessage);
                              virtual ID_First+ID_Fanfare;
      PROCEDURE SoundAlarm    (VAR Msg : TMessage);
                              virtual ID_First+ID_Alarm;
      PROCEDURE SoundGlass    (VAR Msg : TMessage);
                              virtual ID_First+ID_Glass;
      PROCEDURE SoundCough    (VAR Msg : TMessage);
                              virtual ID_First+ID_Cough;
      PROCEDURE SoundSealion  (VAR Msg : TMessage);
                              virtual ID_First+ID_Sealion;
      PROCEDURE SoundSting    (VAR Msg : TMessage);
                              virtual ID_First+ID_Sting;
      END;

PROCEDURE TSoundApp.InitMainWindow;
BEGIN
   MainWindow := New (PSoundWindow, Init (NIL, 'WAV File Playback with Turbo
Pascal for Windows'));
   END;

CONSTRUCTOR TSoundWindow.Init (AParent : PWindowsObject; ATitle : PChar);
CONST
   BXHeight = 80;
   BYHeight = 60;
   BXSpace = 20;
   BYSpace = 20;
   BXWidth = 120;
   BYWidth = 80;

VAR
   LambWFile : FILE;
   Dummy     : WORD;
```

```
BEGIN
   TWindow.Init(AParent,ATitle);
{
 * Specify size and position of window
}
   Attr.x := 50;
   Attr.y := 50;
   Attr.w := 580;
   Attr.h := 320;
{
 * Specify sizes and positions of individual buttons
}
   BSysStart := New (PButton,Init (@Self, ID_SysStart,
                     'WAV File Playback using Turbo Pascal for Windows',
                      10, 10, 550, 30, TRUE));
   BGlass   := New (PButton,Init (@Self, ID_Glass,'Glass',
                    BXHeight,
                    BYHeight,
                    BXWidth,
                    BYWidth,
                    TRUE));
   BCough := New (PButton,Init (@Self, ID_Cough,'Cough',
                    BXHeight+1*(BXWidth+BXSpace),
                    BYHeight,
                    BXWidth,
                    BYWidth,
                    TRUE));
   BFanfare:= New (PButton,Init (@Self, ID_Fanfare,'Fanfare',
                    BXHeight+2*(BXWidth+BXSpace),
                    BYHeight,
                    BXWidth,
                    BYWidth,
                    TRUE));
   BSealion:= New (PButton,Init (@Self, ID_Sealion,'Seal',
                    BXHeight+0*(BXWidth+BXSpace),
                    BYHeight+1*(BYWidth+BYSpace),
                    BXWidth,
                    BYWidth,
                    TRUE));
   BLamb   := New (PButton,Init (@Self, ID_Lamb,'Lamb',
                    BXHeight+1*(BXWidth+BXSpace),
                    BYHeight+1*(BYWidth+BYSpace),
                    BXWidth,
                    BYWidth,
                    TRUE));
   BAlarm  := New (PButton,Init (@Self, ID_Alarm,'Alarm',
                    BXHeight+2*(BXWidth+BXSpace),
                    BYHeight+1*(BYWidth+BYSpace),
                    BXWidth,
                    BYWidth,
                    TRUE));
   BSting  := New (PButton,Init (@Self, ID_Sting,
                    '(C) 1992 Abacus - Author: Axel Stolz',
```

```
                    10, 250, 550, 30, TRUE));

{
 * The Lamb.WAV file loads into memory after window initialization, then
 * the application plays the file. You have the option of stopping a sound
 * in memory, but controlling a resource is much easier.
}
   Assign(LambWFile,'\abacus\sbbook\wav\Lamb.wav');
   Reset(LambWFile,1);
   BlockRead(LambWFile,LambArray,FileSize(LambWFile),Dummy);
   Close(LambWFile);
   END;

PROCEDURE TSoundWindow.SoundSysStart (VAR Msg : TMessage);
{
 * This procedure demonstrates that you can also play the standard
 * system sounds from WIN.INI using SndPlaySound. Normally the [sounds]
 * section must be active in WIN.INI before you can load a WAV file.
}
BEGIN
   { Use WIN.INI entry "SystemStart" }
   SndPlaySoundFile ('SystemStart',SND_ASYNC);
   END;

PROCEDURE TSoundWindow.SoundLamb(VAR Msg : TMessage);

BEGIN
   SndPlaySoundMem (Addr(LambArray),SND_ASYNC OR SND_MEMORY);
   END;

PROCEDURE TSoundWindow.SoundFanfare(VAR Msg : TMessage);
{
 * This demonstrates SndPlaySound using WAV files described in a resource.
 * You can create a WAV resource using a resource script and the resource
 * compiler (RC.EXE).
}

VAR
   WavResPtr  : Pointer;
   WavHandle,
   WavResInfo : THandle;
BEGIN
   WavResInfo := FindResource(HInstance,'Fanfare','WAVE');
   IF Not(WavResInfo>0) THEN
      Halt(1); { Error: Resource not found }
   WavHandle := LoadResource(Hinstance,WavResInfo);
   IF Not(WavHandle>0) THEN
      Halt(2);  { Error: No handle found }
   WavResPtr := LockResource(WavHandle);
   IF (WavResPtr <> NIL) THEN BEGIN
      SndPlaySoundMem (WavResPtr,SND_ASYNC OR SND_MEMORY);
{
 * The following function call releases the resource places in memory
 * by LockResource. Remember that you should only activate this line when
```

```
 * sound is enabled by the SND_SYNC flag, otherwise a general error may
 * occur in Windows, which may cause a crashed sound (remember that a
 * resource is not directly accessible in memory).
      UnlockResource(WavHandle)
}
      END;
   END;

PROCEDURE TSoundWindow.SoundAlarm(VAR Msg : TMessage);
{
 * Demonstration of SND_LOOP parameter flag
}
BEGIN
   SndPlaySoundFile ('\abacus\sbbook\wav\Alarm.wav',SND_ASYNC OR SND_LOOP);
   END;

PROCEDURE TSoundWindow.SoundGlass(VAR Msg : TMessage);
BEGIN
   SndPlaySoundFile ('\abacus\sbbook\wav\Glass.wav',SND_ASYNC);
   END;

PROCEDURE TSoundWindow.SoundCough(VAR Msg : TMessage);
BEGIN
   SndPlaySoundFile ('\abacus\sbbook\wav\Cough.wav',SND_ASYNC);
   END;

PROCEDURE TSoundWindow.SoundSealion(VAR Msg : TMessage);
BEGIN
   SndPlaySoundFile ('\abacus\sbbook\wav\Sealion.wav',SND_ASYNC);
   END;

PROCEDURE TSoundWindow.SoundSting(VAR Msg : TMessage);
BEGIN
   SndPlaySoundFile ('\abacus\sbbook\wav\Sting.wav',SND_ASYNC);
   END;

VAR
   SoundApp : TSoundApp;

BEGIN
   SoundApp.Init('WAV File Playback using Turbo Pascal for Windows');
   SoundApp.Run;
   SoundApp.Done;
   END..i).Program listings:TPWSOUND.PAS;.i).Turbo Pascal for Windows;
```

5.2.7 Sound output for Windows using Borland C++

Sound for Windows using C

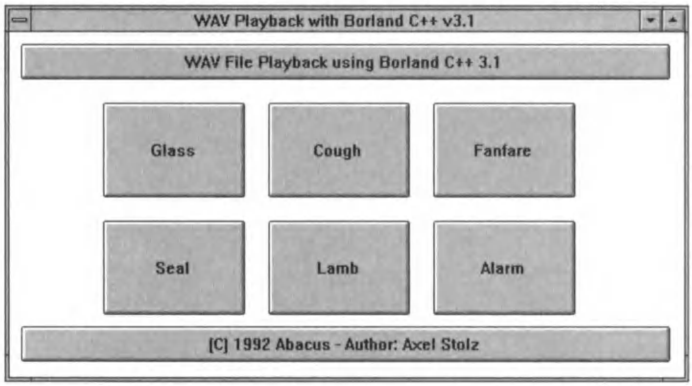
It's also easy to use Borland C++ 3.1 to produce WAV sound output.

The MMSYSTEM.H header file provided with Borland C++ 3.1 contains the necessary sound function. To use functions from a DLL in your own programs, you must add an IMPORTS section to your

project's module definition (DEF) file or link a static LIB file to your project.

CPPSOUND

The ObjectWindows Library (OWL) from Borland allows the C application CPPSOUND.CPP to be almost identical to the TPWSOUND application, in its source code as well as in its resulting user environment.



CPPSOUND window

The program is again equipped with eight buttons.

Sounds from WIN.INI

The title bar button obtains its sound data from the WIN.INI file. This SoundSysStart() function is called using the Systemstart parameter.

sndPlaySound() automatically searches WIN.INI for the specified sound. For the title bar button, this is the same sound that is played back whenever you start Windows.

By using the Sound application from the Windows Control Panel, you can determine which sound is emitted when Windows is started. If you haven't specified a sound, the system default sound is used.

The C version of this demo program doesn't include the playback of a sample in memory through the use of an array. In the TPWSOUND application, the SoundLamb procedure performed this task.

Now you can add various sounds to your C programs and experiment as much as you want.

```
// ***************************************************************************
// * Demonstration application for MMSystem module, Borland C++ 3.1 (WIN)   *
// ***************************************************************************
// *                     Copyright (C) 1992 ABACUS                           *
// *                        Author : Axel Stolz                              *
// ***************************************************************************
// * This application plays WAV files using the SndPlaySound statement as  *
// * used in the MMSYSTEM.DLL.                                               *
// ***************************************************************************

#include <owl.h>
#include <window.h>
#include <button.h>

// Header file for accessing MMSYSTEM.DLL file
#include "mmsystem.h"

//
// * Get constants for different buttons
//

const WORD ID_SysStart    = 101;
const WORD ID_Glass       = 102;
const WORD ID_Cough       = 103;
const WORD ID_Fanfare     = 104;
const WORD ID_Sealion     = 105;
const WORD ID_Lamb        = 106;
const WORD ID_Alarm       = 107;
const WORD ID_Sting       = 108;

class TSoundApp : public TApplication
{
public:
  TSoundApp(LPSTR AName, HINSTANCE hInstance, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int nCmdShow)
    : TApplication(AName, hInstance, hPrevInstance, lpCmdLine, nCmdShow) {};
  virtual void InitMainWindow();
};

class TSoundWindow : public TWindow
{
public:
  TSoundWindow(PTWindowsObject AParent, LPSTR ATitle);
// Assign ID constants to button procedures
  virtual void SoundSysStart(RTMessage Msg) = [ID_FIRST + ID_SysStart];
  virtual void SoundGlass(RTMessage Msg)    = [ID_FIRST + ID_Glass];
  virtual void SoundCough(RTMessage Msg)    = [ID_FIRST + ID_Cough];
  virtual void SoundFanfare(RTMessage Msg)  = [ID_FIRST + ID_Fanfare];
  virtual void SoundSealion(RTMessage Msg)  = [ID_FIRST + ID_Sealion];
  virtual void SoundLamb(RTMessage Msg)     = [ID_FIRST + ID_Lamb];
  virtual void SoundAlarm(RTMessage Msg)    = [ID_FIRST + ID_Alarm];
  virtual void SoundSting(RTMessage Msg)    = [ID_FIRST + ID_Sting];
};
```

```
TSoundWindow::TSoundWindow(PTWindowsObject AParent, LPSTR ATitle) :
  TWindow(AParent, ATitle)
{
   const BYTE BXHeight = 80;
   const BYTE BYHeight = 60;
   const BYTE BXSpace = 20;
   const BYTE BYSpace = 20;
   const BYTE BXWidth = 120;
   const BYTE BYWidth = 80;

//
// * Specify window size and position
//
  Attr.X = 50;
  Attr.Y = 50;
  Attr.W = 580;
  Attr.H = 320;

//
// * Specify button sizes and positions
//
  new TButton(this, ID_SysStart,
    "WAV File Playback using Borland C++ 3.1",
    10, 10, 550, 30, FALSE);
  new TButton(this, ID_Glass, "Glass",
    BXHeight,
    BYHeight,
    BXWidth,
    BYWidth,
    FALSE);
  new TButton(this, ID_Cough, "Cough",
    BXHeight+1*(BXWidth+BXSpace),
    BYHeight,
    BXWidth,
    BYWidth,
    FALSE);
  new TButton(this, ID_Fanfare, "Fanfare",
    BXHeight+2*(BXWidth+BXSpace),
    BYHeight,
    BXWidth,
    BYWidth,
    FALSE);
  new TButton(this, ID_Sealion, "Seal",
    BXHeight+0*(BXWidth+BXSpace),
    BYHeight+1*(BYWidth+BYSpace),
    BXWidth,
    BYWidth,
    FALSE);
  new TButton(this, ID_Lamb, "Lamb",
    BXHeight+1*(BXWidth+BXSpace),
    BYHeight+1*(BYWidth+BYSpace),
    BXWidth,
    BYWidth,
    FALSE);
```

```
  new TButton(this, ID_Alarm, "Alarm",
    BXHeight+2*(BXWidth+BXSpace),
    BYHeight+1*(BYWidth+BYSpace),
    BXWidth,
    BYWidth,
    FALSE);
  new TButton(this, ID_Sting,
    "(C) 1992 Abacus - Author: Axel Stolz",
    10, 250, 550, 30, FALSE);

}

void TSoundWindow::SoundSysStart(RTMessage)
{
//
// * This demonstrates that you can access WIN.INI system sounds for
// * playback. Normally the [sounds] section must be active in WIN.INI
// * before you can load a WAV file.

//
  sndPlaySound("systemstart",SND_ASYNC);
}

void TSoundWindow::SoundGlass(RTMessage)
{
 sndPlaySound("\\abacus\\sbbook\\wav\\Glass.wav",SND_ASYNC);
}

void TSoundWindow::SoundCough(RTMessage)
{
 sndPlaySound("\\abacus\\sbbook\\wav\\Cough.wav",SND_ASYNC);
}

void TSoundWindow::SoundFanfare(RTMessage)
{
 sndPlaySound("\\abacus\\sbbook\\wav\\Fanfare.wav",SND_ASYNC);
}

void TSoundWindow::SoundSealion(RTMessage)
{
 sndPlaySound("\\abacus\\sbbook\\wav\\Sealion.wav",SND_ASYNC);
}

void TSoundWindow::SoundLamb(RTMessage)
{
 sndPlaySound("\\abacus\\sbbook\\wav\\Lamb.wav",SND_ASYNC);
}

void TSoundWindow::SoundAlarm(RTMessage)
{
//
// * Demonstrates SND_LOOP parameter flag
//
 sndPlaySound("\\abacus\\sbbook\\wav\\Alarm.wav",SND_ASYNC || SND_LOOP);
```

```
}

void TSoundWindow::SoundSting(RTMessage)
{
 sndPlaySound("\\abacus\\sbbook\\wav\\Sting.wav",SND_ASYNC);
}

void TSoundApp::InitMainWindow()
{
  MainWindow = new TSoundWindow(NULL, Name);
}

int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
  LPSTR lpCmdLine, int nCmdShow)
{
  TSoundApp TestApp("WAV Playback with Borland C++ v3.1",
    hInstance, hPrevInstance, lpCmdLine, nCmdShow);
  TestApp.Run();
  return TestApp.Status;
}
```

Here's the module definition (DEF) file you'll need for compiling CPPSOUND.C.

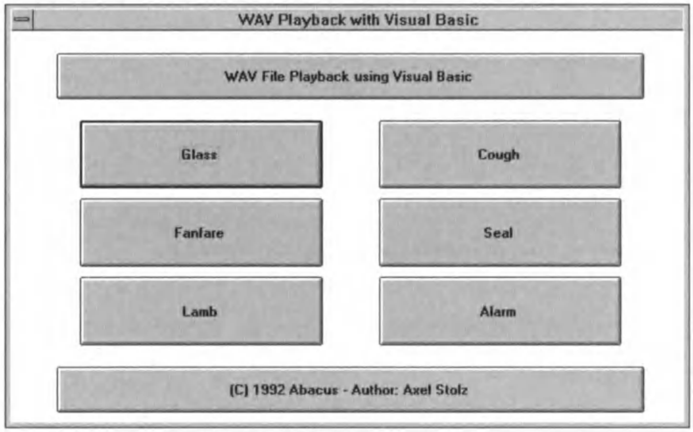
```
EXETYPE WINDOWS
CODE PRELOAD MOVEABLE DISCARDABLE
DATA PRELOAD MOVEABLE MULTIPLE
HEAPSIZE 4096
STACKSIZE 5120
```

5.2.8 Sound playback using Visual Basic

Sound using Visual Basic

Sound playback through Visual Basic under Windows is as easy as with Turbo Pascal for Windows.

The project VCSOUND.MAK illustrates how Visual Basic is used to implement sound samples.



VBSOUND window

The sound flags are defined as global constants in the GLOBAL.BAS module. Their properties and uses are identical to those described in the section on Turbo Pascal.

The SndPlaySound procedure from MMSYSTEM.DLL is also linked to the project in this global module.

The first parameter for this function is a string containing the name of the WAV file that must be played. As in the Turbo Pascal for Windows version, this string may be the name of one of the sounds contained in WIN.INI.

The top button of VBSOUND1.FRM might, for example, call the SystemStart sound from the WIN.INI file. All sounds for the other buttons are loaded from their corresponding WAV files.

Resources and Visual Basic

Unlike Turbo Pascal for Windows, Visual Basic doesn't allow sound samples to be played back from resources.

Sounds from a WAV file

For example, consider the sub-procedure for the Alarm button. Similar to the other sub-procedures, it consists of only a single line, the SndPlaySound function call. The filename ALARM.WAV is specified as the desired WAV file, along with the appropriate search path.

Remember that you must modify the search paths for these sounds throughout the application if you haven't installed the sounds in the ABACUS\SBBOOK\WAV directory.

The Alarm sample differs from the other sounds because not only the SND_ASYNC but also the SND_LOOP sound flag has been set with this function call. These two flags are linked using the logical OR operator.

Sounds from WIN.INI

Another way of playing back sound samples uses the sound settings stored in WIN.INI. The title bar button in this program uses this approach. This button activates the TitleSound_Click Sub routine.

This routine calls SndPlaySound with the SystemStart parameter. Instead of a variable containing a WAV filename, this is a sound sample contained in WIN.INI.

So the exact sample that this function call plays back depends on the setting you've specified in your Windows system settings. You'll find more information about this in Chapter 3.

With this information, you're ready to enter the world of digitized sound under Visual Basic. This method offers as many possibilities as Turbo Pascal.

By using digitized sound you can add an entirely new dimension to your programs.

The VBRUN100.DLL dynamic link library is needed for the compiled program on the companion diskette to operate properly.

Here are the Visual Basic listings in text form.

```
' GLOBAL.BAS
' ***************************************************************************
' * Microsoft Windows 3.0 + MM Extensions / Windows 3.1 Sound Interface *
' ***************************************************************************
' *                    Copyright (C) 1992 Abacus                            *
' *                             Authors :                                   *
' *                        Thorsten Petrowski                               *
' *                            Axel Stolz                                   *
' ***************************************************************************
' *               WAV file playback using Visual Basic                      *
' ***************************************************************************


Global Const SND_SYNC = 0         ' Play synchronous sound (default)
Global Const SND_ASYNC = 1        ' Play asynchronous sound
Global Const SND_NODEFAULT = 2    ' Do not use INI sounds if
                                  ' no .WAV files can be found
Global Const SND_MEMORY = 4       ' Pointer points to data in memory
Global Const SND_LOOP = 8         ' Repeat sound until next SndPlaySound
Global Const SND_NOSTOP = 10      ' Do not interrupt playing sound
```

```
Declare Sub SndPlaySound Lib "MMSystem" (ByVal Filename As String,
                                         ByVal Flags As Integer)
' VBSOUND1.FRM
' ***************************************************************************
' *   Demonstration application using the MMSystem DLL from Visual Basic   *
' ***************************************************************************
' *                    Copyright (C) 1992 Abacus                            *
' *                       Author : Axel Stolz                               *
' ***************************************************************************
' *  The SndPlaySound procedure, provided by the MMSystem DLL, allows easy *
' *                  playback of WAV files.                                 *
' ***************************************************************************

Sub FanfareSound_Click ()
    Call SndPlaySound("\abacus\sbbook\wav\fanfare.wav", SND_ASYNC)
End Sub

Sub AlarmSound_Click ()
    Call SndPlaySound("\abacus\sbbook\wav\alarm.wav", SND_ASYNC Or SND_LOOP)
End Sub

Sub SeaLionSound_Click ()
    call SndPlaySound("\abacus\sbbook\wav\sealion.wav", SND_ASYNC)
End Sub

Sub StingSound_Click ()
    Call SndPlaySound("\abacus\sbbook\wav\sting.wav", SND_ASYNC)
End Sub

Sub SysStartSound_Click ()
    Call SndPlaySound("systemstart", SND_ASYNC)
End Sub

Sub GlassSound_Click ()
    Call SndPlaySound("\abacus\sbbook\wav\glass.wav", SND_ASYNC)
End Sub

Sub LambSound_Click ()
    Call SndPlaySound("\abacus\sbbook\wav\lamb.wav", SND_ASYNC)
End Sub

Sub CoughSound_Click ()
    Call SndPlaySound("\abacus\sbbook\wav\cough.wav", SND_ASYNC)
End Sub
```

5.3 Programming the FM Voices

FM voices

In this section we'll discuss how to program the Sound Blaster card's FM voices. We'll focus on the use of the SBFMDRV.COM driver in Turbo Pascal 6.0 and Borland C++.

However, to give you an overview of how the FM voices are produced, we'll discuss the process of FM synthesis.

5.3.1 FM synthesis theory

FM synthesis

It's easier to program your own instrument voices if you understand how your sound card's FM chip operates.

The synthesizer chip of your Sound Blaster card has 18 operators. Generally two such operators are required to produce one instrument voice. This means that the card is capable of producing a melody with a maximum of nine different voices.

You're probably wondering why the Sound Blaster card supposedly has 11 voices. This is because FM chips are also capable of creating percussion sounds in addition to melodic voices. These percussion sounds usually require only one operator.

However, you should understand this more clearly after we discuss the different operating modes of the Sound Blaster card.

FM operating mode 1

In its first mode, the Sound Blaster card can produce nine different melodic voices. This means that two operators are used for each voice. However, this mode doesn't support the use of percussion sounds.

FM operating mode 2

The second operating mode has only six melodic voices, which require the first 12 operators.

In addition to these six voices, this mode also provides five percussion sounds. The Bass Drum sound uses two operators and the remaining four (Hi Hat, Tom Tom, Snare Drum, and Top Cymbal) each use a single operator.

So all 18 operators are fully used, providing 11 different voices. This is the most common operating mode of the FM chip.

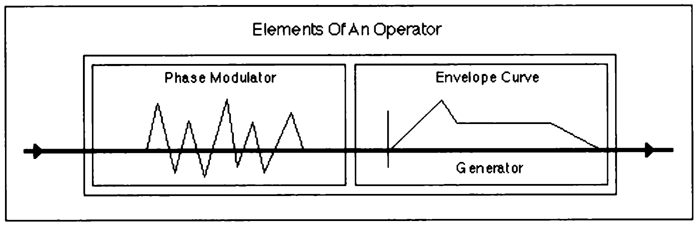
FM operating mode 3

The third operating mode is rarely used because its function is almost entirely replaced by the digital sound channel. This mode is also known as "Speech synthesis."

Speech synthesis allows all 18 operators to be switched together so an extremely complex signal wave, which imitates speech, is produced. However, unlike the playback of sound samples, this is a difficult task.

Structure of an operator

An operator can be divided into two basic elements. The first is its phase modulator and the second is its envelope generator.



Simplified illustration of an operator

As we mentioned in the section on music theory, a tone primarily consists of two quantities:

- Frequency determines the pitch of the tone.
- Amplitude determines its loudness.

In an operator, the phase modulator determines the frequency and the envelope generator determines the loudness.

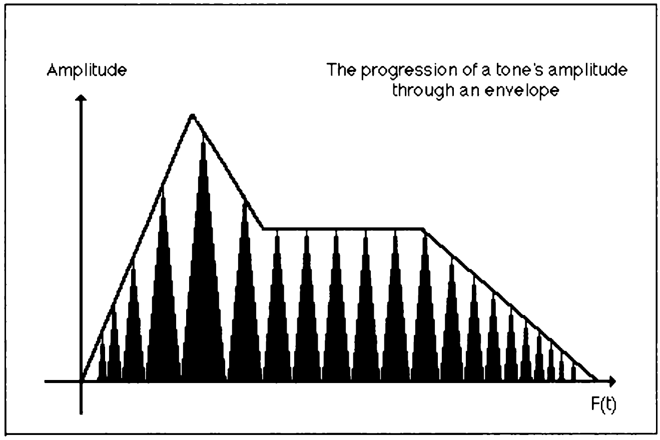
Envelope

Earlier we discussed frequency. Now we must explain envelopes.

The envelope "envelops" a tone's frequency in reference to its amplitude. It determines the progression of a tone's loudness.

Different musical instruments have different envelopes, which contribute to their sound qualities on combination with the instrumental timbres.

For example, imagine a single note played on a piano. The tone rises quickly to its maximum amplitude, falls back slightly, remains at a certain amplitude for a short period, and then slowly fades off. By pressing the sustain pedal you can cause the tone to sound longer.



Determining the progression of a tone's amplitude

Envelopes can be used to program such loudness progressions. This can be done by simply defining a few important parameters. These parameters are Attack, Decay, Sustain and Release.

Attack

The attack or rise of a tone determines the amount of time required for the tone to reach its maximum amplitude. A piano or harpsichord has a faster attack than a violin, for example.

Decay

The decay determines the amount of time required for the tone to fall from its maximum loudness to an amplitude level that is then sustained for a certain period.

Sustain

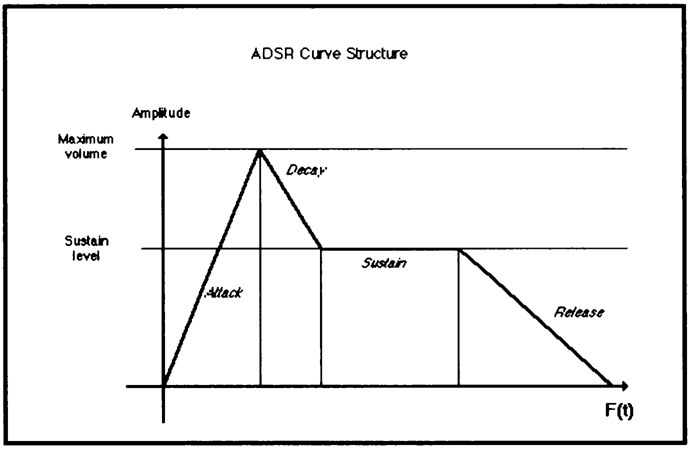
The sustain is the amplitude at which the tone is emitted for a certain amount of time before it finally fades away.

Release

The release is a second fade; the tone actually fades away completely until the amplitude of its pulses has reached zero. The tone's release determines the time its amplitude needs to reach zero.

NOTE

The envelope generator is also called the ADSR generator. This term is formed by the first letter of each of the attributes.

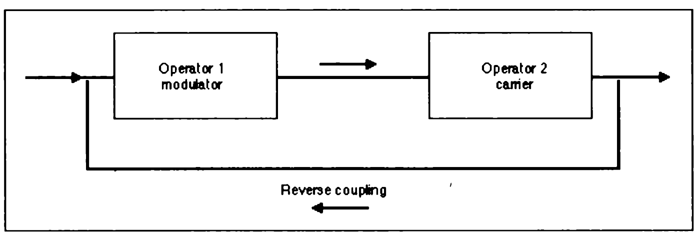


Typical structure of an ADSR curve

So an operator produces a sound wave that's determined through the settings of the phase modulator and the ADSR generator.

The principle of FM synthesis

During FM synthesis, two operators are placed back to back. The output signal of the first operator is used as the input signal of the second. So the first operator acts as the modulator, while the second acts as the carrier.



Operator sequence during FM synthesis

The carrier is responsible for the base frequency of the emitted tone, while the modulator determines its overtones.

As we mentioned, overtones determine much of the timbre (sound quality) of a tone. Their frequency and amplitude sequence give a tone its identity. You can influence these factors by modifying the modulator frequency and the modulator envelope curve.

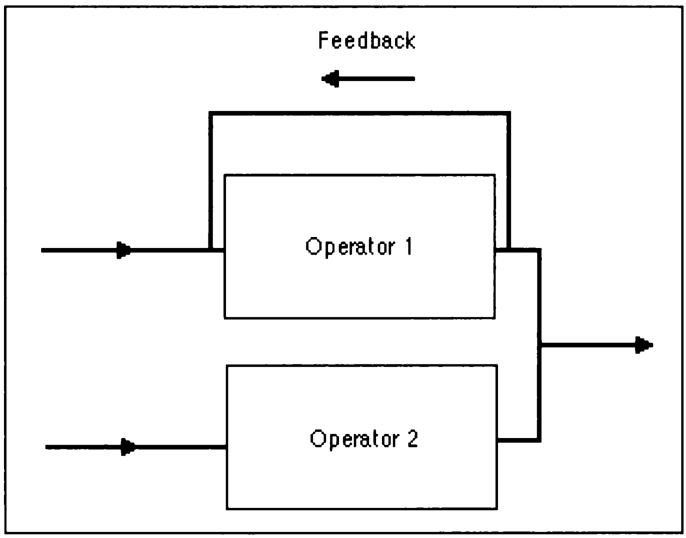
There are several other parameters that determine the exact function of an operator. The settings that are important for using

instrument voices are described in more detail in the explanation of the Sound Blaster Instrument format.

Another effect can be created by connecting the combined output signal of two operators to the modulator's input through a feedback loop.

These settings allow you to create very complex sounds by using only two operators.

However, operators can also be used in a parallel arrangement. This type of synthesis is known as additive synthesis.



Operator sequence during additive synthesis

During additive synthesis, both operators produce a sine wave. To produce the final output signal, these two waves are then added to each other. So additive synthesis is a type of Fourier synthesis, only on a very small scale.

Since it doesn't include a modulator that's capable of producing a series of overtones in a single pass, this type of synthesis cannot produce signals that are as complex as the signals created through FM synthesis. Therefore, instrument voices created through additive synthesis sound much more artificial than those created using FM synthesis.

However, which type of synthesis is suitable for a given application depends on the type of tone you want to produce. You

must experiment with the two methods to understand this. An instrument editor that allows you to test a newly created instrument is extremely helpful.

Instrument editors are available through Creative Labs and shareware sources for editing existing instrument sounds and creating new ones. Although producing the sound you want requires a lot of time and experimentation, you'll discover many interesting sounds in the process.

We'll offer one small warning in creating your own instruments for FM synthesizer. FM instrument generation is often a complex and frustrating task. We suggest that if you do any experimenting in instrument editing, you take an existing FM instrument sound and tweak that, rather than starting from scratch.

5.3.2 SBI format

Since several parameters are required to fill the registers of the FM chip for each operator, Creative Labs has introduced the SBI format. SBI is an abbreviation for "Sound Blaster Instrument" and contains, in addition to the operator parameters, other useful information. Each Sound Blaster Instrument is $33 (51) bytes long.

Bytes $00 - $03 (0 - 3)

File ID

These bytes contain the text "SBI", ending with byte $1A. By checking the values of these four bytes you can verify whether the file is an SBI file.

Bytes $04 - $23 (4 - 35)

Instrument name

These bytes contain the instrument's name, which can contain a maximum of 32 bytes, including the terminating zero byte.

Byte $24 (36)

This byte contains a series of encoded flags.

Modulator sound properties

When set, bit 7 indicates that the modulator must produce an amplitude vibrato. This means that the tone's amplitude is alternately raised and lowered by one decibel during playback, which results in a vibrato effect (amplitude vibrato).

When bit 6 is set, it indicates that the modulator must produce a frequency vibrato. This means that the tone's frequency is alternately raised and lowered by 1/100th of a half-step, which results in a different vibrato effect (frequency vibrato).

When bit 5 is set, the tone must be maintained at the specified sustain level as long as it's switched on. If the bit isn't set, the tone immediately goes from Decay to Release, regardless of the tone's duration.

When bit 4 is set, the duration of a tone depends on its frequency. This means that higher tones are of shorter duration than longer tones (Key Scale Rate). This allows the sound of several types of instruments to be reproduced more accurately. So this function shortens the ADSR curve of higher tones.

Bits 0 - 3 specify the value of a frequency multiplier. This multiplier is applied to the operator's base frequency. The four bytes permit the use of 16 different multipliers. The multiplier 0.5 corresponds to the bit value 0. This means that the emitted tone will have only half the frequency that was specified by the base tone. So the resulting tone is exactly one octave lower.

For a bit value of 1, the multiplier is exactly 1. So the emitted tone is of the same frequency as the base tone. A bit value of 2 corresponds to a multiplier of 2, so the emitted tone is one octave above the base frequency.

However, this sequence doesn't continue sequentially. So we've included the following table, which shows the corresponding multipliers:

Bit value	Multiplier	Bit value	Multiplier
0	0.5	1	1.0
2	2.0	3	3.0
4	4.0	5	5.0
6	6.0	7	7.0
8	8.0	9	9.0
10	10.0	11	10.0
12	12.0	13	12.0
14	15.0	15	15.0

By using the multiplier rates shown above, as well as the rule "2x frequency = one octave higher", you'll be able to calculate the effects of different settings for the bits 0 - 3.

Byte $25 (37)

Carrier sound properties

This byte is identical to byte $24, except that these data apply to the carrier instead of the modulator.

Byte $26 (38)

This byte contains two different pieces of information.

Modulator volume

Bits 7 and 6 represent a factor by which the loudness of an emitted tone must be changed whenever its frequency changes. This means that higher tones are played more quietly than low tones.

The bit values 0, 1, 2, and 3 correspond to a decrease of the tone's amplitude of 0, 1.5, 3, and 6 decibels per octave (Key Scale Level), respectively.

Bits 5 - 0 specify the operator's overall loudness. These bits permit values from 0 to 63 to be used. Similar to the decibel scale, 0 is the loudest.

Byte $27 (39)

Carrier volume

This byte is identical to byte $26 of the modulator, except that the parameters apply to the carrier operator.

Byte $28 (40)

Modulator attack/decay

This byte contains the data that determines the modulator's attack and decay.

Bits 7 - 4 contain the values for the attack rate. The value 0 is the slowest and 15 is the fastest.

The bits 3 - 0 contain the values for the decay rate. The value 0 is the slowest and 15 is the fastest.

Byte $29 (41)

Carrier attack/decay

This byte is identical to byte $28 for the modulator, except that the parameters apply to the carrier operator.

Byte $2A (42)

Modulator sustain/release

This byte contains the data needed for the modulator's sustain and release.

Bits 7 - 4 contain the values for the sustain level. 0 is the loudest value and 15 is the quietest.

Bits 3 - 0 contain the values for the release rate. The value 0 is the slowest and 15 is the fastest.

Byte $2B (43)

Carrier sustain/release

This byte is identical to byte $2A for the modulator, except that the parameters apply to the carrier operator.

Byte $2C (44)

Modulator waveform

This byte determines how strongly the modulator signal must be distorted. Values from 0 (no distortion) to 3 (high distortion) are permitted, and bits 0 and 1 are used. The other bits aren't used.

Byte $2D (45)

Carrier waveform

This byte describes the distortion of the carrier operator signal. The values are defined the same way as those in byte $2C for the modulator.

Byte $2E (46)

Synthesis mode and phase shifting

This byte determines how the two operators must be combined and how the feedback loop is configured.

Bits 3 - 1 specify a value between 0 and 7, which determine the phase shift that occurs because of the feedback loop. Depending on the bit values, the phases are shifted by 0, Pi/16, Pi/8, Pi/4, Pi/2, Pi, Pi*2, or Pi*4.

When bit 0 is set, both operators are configured in parallel. This means that the signal is formed through additive synthesis. If bit 0 contains the value zero, the signal is formed through FM synthesis.

The remaining bits are unused.

Bytes $2F - $33 (47 - 51)

Bytes for future use

These bytes are currently unused, reserved for future expansions.

All these bytes are contained in the SBI format. Since their explanations are only theoretical, you don't need to know every detail of every bit.

If you have a good instrument editor, you don't have to worry about how this information is stored. However, knowing the effects of different settings will help you create your own instruments.

5.3.3 CMF file format

The special music files for your Sound Blaster card can usually be identified by their .CMF filename extension. These files are always stored in the Creative Music File format. This file format, like the Creative Voice File Format, was developed by Creative Labs.

A CMF file is divided into three main blocks:

- Header
- Instrument block
- Music block

The special characteristic of CMF is that its music block is identical to the MIDI format specified by the International MIDI Association. This means that the music information stored in CMF files follows MIDI specifications.

The only difference is that a CMF file is also equipped with a special header, and instrument information for the sound card's FM chip.

CMF header

The header of a CMF file contains a lot of information about the file's structure. Its length depends on the size of the individual file. Information that's two bytes long is stored in the format low-byte/high-byte, corresponding to the WORD data type.

Bytes $00 - $03 (0 - 3)

File ID

These bytes must contain the value "CTMF" as ASCII text. These characters indicate that the file corresponds to the CMF format.

Bytes $04 - $05 (4 - 5)

Version number

These two bytes contain the sub and main digits of the CMF format version number.

Bytes $06 - $07 (6 - 7)

Instrument offset

These bytes contain the offset address of the instrument address, measured from the start of the file. This address is important when the instruments must be read from a CMF file so this information can be passed to the FM chip.

Bytes $08 - $09 (8 - 9)

Music offset

These bytes contain the offset address of the music data, which is also measured from the start of the file. This address is needed to determine where the header and instrument data end and where the music data begins.

Bytes $0A - $0B (10 - 11)

Ticks per quarter

The values stored in these bytes determine how many timer ticks must represent one quarter note in the piece of music. We'll explain the function of these ticks later when we discuss a corresponding driver function.

Bytes $0C - $0D (12 - 13)

Clock ticks per second

These two bytes contain the number of interrupt calls that must be sent from the system timer interrupt 0 to the music driver each second. The default value is 96 Hz ($60), which means that the FM driver is called by the system 96 times every second.

Bytes $0E - $ 0F (14 - 15)

CMF title

These bytes contain the offset address of the title for the music data, which is again measured from the start of the file. The title is stored as an ASCII text, which must be ended with a zero byte. If this offset address is zero, then the piece doesn't have a title.

Bytes $10 - $11 (16 - 17)

Author name

These bytes contain the offset address of an ASCII text, which states the author's name. This text must also be ended with a zero byte. If the address value is zero, the file doesn't contain an author name.

Bytes $12 - $13 (18 - 19)

CMF remarks

This value is the offset address of an ASCII text used for comments about the piece of music contained in this file. The text must also be ended with a zero byte. If the address value is zero, the file doesn't contain any remarks.

Bytes $14 - $23 (20 - 35)

Channel table

Since CMF corresponds to the MIDI format, it also supports 16 channels. These 16 bytes are used to indicate which of the 16 channels are being used by the data in the file. When a byte

contains the value 1, the channel is used; 0 indicates an unused channel.

Bytes $24 - $25 (36 - 37)

Number of instruments

These two bytes contain the number of instruments used within the piece of music. The maximum number of instruments that can be used is 128.

Bytes $26 - $27 (38 - 39)

Tempo

The value contained in these two bytes determines the basic tempo of the music contained in the file.

Bytes $28 - ... (40 - ...)

Text bytes

Starting with byte 28, the header may contain the previously described information (i.e., the title, author name, and comments about the piece).

These texts are followed by the instrument block. Since the length of these texts isn't predetermined, you must obtain the starting location of the instrument data from bytes $06 and $07 of the header.

The data for each instrument consists of 16 bytes. This corresponds to the 16 bytes that are required to provide the necessary values for the FM chip. These bytes also correspond to the 16 bytes, starting with number $24, in the SBI format.

5.3.4 How SBFMDRV.COM operates

A driver for the Sound Blaster card's FM voices is included in the Sound Blaster package. This program is called SBFMDRV.COM.

First, you must start SBFMDRV.COM so it will become resident in memory as a TSR program. Once you've installed it, other programs, such as PLAYCMF.EXE by Creative Labs, will be able to access it by calling the interrupt address at which the driver has installed itself.

In the following section, we'll explain how to access this driver from within your Turbo Pascal or C programs. This enables these programs to play CMF music files in the background.

However, first let's discuss the driver's functions.

The number of the desired function is written to the BX register before the interrupt is called.

Function 0 (BX=0): Get driver version number

This function returns the version number of the SBFMDRV driver, which is currently installed in memory. Register AX will contain the main and subversion numbers after the function call.

Get driver version	
Input	BX = 00
	AH = Main number
	AL = Sub-number
Remarks	none

Function 1 (BX=1): Set status byte

This function is used to tell the driver the location to which it must write its status information. While a CMF file is being played back, this status byte contains a value that doesn't equal zero. Once music data isn't being sent and the piece has ended, this byte is set back to zero.

Set status byte	
Input	BX = 01
	DX:AX = Status address
Output	none
Remarks	none

Function 2 (BX=2): Set instruments

This function sets the instrument voices for the piece (i.e., the appropriate FM chip parameters). You need this function when you want to play back a CMF file, for example.

Function 2 is used to pass the instrument data contained in the CMF file to the FM chip registers.

In register CX you must specify the number of instruments that will be used, and in register pair DX:AX you must specify the segment:offset address of the instrument data.

Since the CMF format corresponds to the standard MIDI format, it also recognizes the Program-Change command.

When this MIDI command is encountered within a set of music data, function 2 is activated.

Set instruments	
Input	BX = 02
	CX = Number of instruments
	DX:AX = Start of instrument data
Output	none
Remarks	none

Function 3 (BX=3): Set system clock rate

This function sets the rate of the system timer "Timer 0" back to the specified value after the music file has been played.

This value, which must be entered in register AX, is calculated from the equation:

```
1193180/system_clock_rate_in_Hertz
```

This means that, to obtain the value needed for the desired rate, you must divide 1,193,180 by the desired rate; the latter value must be a Hertz unit.

If you don't activate this function, Timer 0 is automatically reset to a frequency of 18.2 Hz.

Set system clock	
Input	BX = 03
	AX = 1193189/System clock rate
Output	none
Remarks	none

Function 4 (BX=4): Set driver clock rate

With this function you can specify the frequency to which the system timer 0 will be set at the beginning of a music file playback.

The value that must be passed to the AX register for this function is calculated in the same way as for function 3, except that you

must divide by the desired driver clock rate in Hertz, instead of the system clock rate.

The appropriate clock rate for a given CMF music file is stored in bytes $0C and $0D (see information on the CMF format).

If you specify a large clock rate, the piece is played faster, and if you select a smaller rate, it's played more slowly. The default clock rate is 96 Hz.

Set driver clock	
Input	BX = 04
	AX = 1193189/Driver clock rate
Output	none
Remarks	none

Function 5 (BX=5): Transpose all notes

This function allows you to transpose a musical score up or down in pitch, by a specified number of half-tone steps. You must specify the number of half-tone steps in the AX register. Positive values shift the pitch of the piece up. Negative values shift the pitch down.

Transpose all notes	
Input	BX = 05
	AX = Half-tone steps
Output	none
Remarks	All notes will be transposed at playback

Function 6 (BX=6): Start playback

Function 6 is needed to start the playback of a music file. The segment:offset address for the music data must be passed to register pair DX:AX with this function call.

If the piece was played successfully and no errors occurred, the value zero is returned in the AX register. If this register contains the value 1, then another piece was being played back.

This procedure sets the status byte to $FF, changes the system clock rate to the desired driver clock rate, and starts the playback by using the timer interrupt.

Start playback	
Input	BX = 06
	DX:AX = Address of music data
Output	AX = 0, no error
	AX = 1, another piece is currently being played back
Remarks	none

Function 7 (BX=7): Stop playback

This function stops the playback of music data, resets timer 0 to the selected system clock rate, and resets the timer interrupt to the original procedure.

If the value zero is returned in register AX, then the stop was successful. If 1 is returned, a piece wasn't being played back.

Stop playback	
Input	BX = 07
Output	AX = 0, no error
	AX = 1, no piece was currently being played back
Remarks	none

Function 8 (BX=8): Initialize driver

This function switches off the FM chip and resets all instrument definitions to their output values. If the initialization was successful, the value zero is returned in the AX register. If the value 1 is returned, then a piece is still being played back.

You should call this function before ending any program that has used the driver.

Initialize driver	
Input	BX = 08
Output	AX = 0, no error
	AX = 1, a piece is still being played back
Remarks	none

Function 9 (BX=9): Pause playback

This function is used to pause a piece that's currently being played back. The status byte value isn't changed by this function.

If register AX contains the value zero following the function call, then the playback has been paused successfully. If the value 1 is returned, a music file wasn't currently being played back.

Pause playback	
Input	BX = 09
Output	AX = 0, no error
	AX = 1, no piece is currently being played back
Remarks	none

Function 10 (BX=10): Continue playback

This function is used to continue the playback of a music file that has been paused using function 9. If this function is executed successfully, the value zero is returned in the AX register. The value 1 indicates that a piece wasn't paused at the time of the function call.

Continue music playback	
Input	BX = 10
Output	AX = 0, no error
	AX = 1, no piece was paused at the time of the function call
Remarks	none

Function 11 (BX=11) Set user-defined function

Since the CMF music format corresponds to the standard MIDI format, it may also contain system exclusive messages.

For your program to process these messages, you must use function 11. The segment:offset address of your own procedure is passed to the driver in the register pair DX:AX. Then this procedure is called each time a system exclusive message is encountered.

Your procedure finds the address of the system exclusive message in the register pair ES:DI. Ensure that your procedure secures the contents of all registers and that it ends with "RETF".

To deactivate your procedure, assign the value zero to DX and AX and call function 11 again.

Set user-defined function	
Input	BX = 11
	DX:AX = Address of user procedure
Output	none
Remarks	none

Now you have an overview of the functions supported by the SBFMDRV.COM driver. In the next section, you'll discover how you can use these functions from within your own programs.

5.3.5 CMF programming using Turbo Pascal 6.0

In this section we'll show you how to use SBFMDRV.COM with a high level language. We'll use Turbo Pascal 6.0.

CMFTOOL.PAS

The center of this example is the CMFTOOL.PAS unit. This unit contains all the functions supported by the driver, except for function 11. Since this function is different for each application, you should create your own for each of your programs, if the function is needed.

However, the other functions are treated the same way as the procedures introduced with the VOCTOOL unit, as listed earlier in this chapter. Also, most of the error messages are the same. So it should be easy to compare and work with the two units.

The CMFTOOL.PAS unit also provides the header of a CMF file in the form of the data structure CMFHeader. This makes the

information contained in the header more accessible to your programs.

Automatic initialization

In the units initialization section, several important functions are performed. First, the global error variable CMFErrStat is set to zero, and CMFSongPaused is set to FALSE.

Then the unit checks whether the driver SBFMDRV.COM is currently resident in memory. The CMFInitDriver function is used to do this.

If this function returns the value TRUE, then the driver has been found and the global variable CMFDriverInstalled is also set to TRUE. If the driver couldn't be located, CMFDriverInstalled receives the value FALSE.

This allows you to determine, at the start of your programs using the CMFTOOL unit, whether the required driver has already been loaded.

Once the driver has been initialized successfully, it receives the address of the global variable CMFStatusByte. Your programs can use this variable at all times to determine the current status of a CMF file playback.

The variable CMFDriverIRQ contains the interrupt number through which the driver can be called. Never change this value; otherwise your program will crash.

Global error messages

Each time an error occurs with the CMFTOOL unit, the global error variable CMFErrStat is set to the corresponding error number, which can then be read by your program.

The program reacts accordingly to errors. However, you must remember to set the error variable back to zero once the error has been processed; otherwise it will retain that number until the next error occurs.

CMFTOOL error numbers

1xx error numbers are reserved for errors that occur during the initialization of the driver.

Error 100

During initialization, error 100 is used when an interrupt, to which the driver was linked, isn't found.

Error 110 Error 110 indicates that the CMFReset function call was unsuccessful because a music file was still in playback.

2xx error numbers are used for errors that may occur while CMF files are being loaded.

Error 200 Error number 200 is used when the specified CMF file couldn't be located. Either an incorrect name was specified, or the file wasn't located at the indicated path.

Error 210 Error number 210 occurs when there isn't enough memory to load the desired CMF file.

This may happen because you've forgotten to set an upper memory limit within your program. Your program then reserves all available memory so a CMF file cannot be loaded.

Your system probably isn't actually out of memory because CMF files are much smaller than sample files.

Error 220 Error number 220 is activated when the loaded file isn't a CMF file. This identity check uses the first 4 bytes of the file's header. These bytes must contain the characters "CTMF".

Error 300 As in the VOCTOOL unit, the error number 300 indicates that an error has occurred while trying to free a memory area. This error usually occurs when you try to free a memory area that wasn't first reserved.

Error 400 Error number 400 indicates that you've tried to pass more than 128 instrument definitions to the driver.

5xx error numbers occur with the playback of CMF data.

Error 500 Error number 500 indicates that a CMF song couldn't be started. This is usually because another song was being played back at the time.

Error 510 Error number 510 is activated when a music data playback couldn't be interrupted. This occurs because a playback wasn't in progress at the time of the function call.

Error 520 Error number 520 indicates that an attempt to pause a playback has failed because a playback wasn't in progress at the time of the function call.

Error 530

Error number 530 indicates that a paused playback couldn't be continued.

CMFTOOL functions and procedures

In the following section we'll list all the functions contained in the CMFTOOL unit that can be accessed through the unit's INTERFACE statement.

Procedure PrintCMFErrMessage

This procedure displays a CMF error on your screen as text. You can use this procedure within the error processing procedures of your own programs because it uses only a simple Write statement.

While you're programming the unit, use this function to help you remember the unit's error numbers.

Function CMFGetSongBuffer

This function performs all the tasks that are needed to load a CMF file into memory. You only need the following parameters to pass to this function:

- A pointer variable which will then point to the loaded CMF data.
- The name of the file that must be loaded in the form of a character string.

The function then tries to open this file and reserve the required memory area. Then the data are read into the reserved memory area.

If the function was successful, it returns the value TRUE. If the value FALSE is returned, you must check the error number contained in CMFErrStat.

Function CMFFreeSongBuffer

This function frees the memory area that has previously been reserved by CMFGetSongBuffer. You must provide the pointer variable identifying the CMF data as a function parameter.

If error number 300 occurs when the function is activated, the function value is FALSE; otherwise it's TRUE.

Function CMFInitDriver

This function searches for the installed driver SBFMDRV. The function searches interrupts $80 through $BF to determine whether the characters "FMDRV" are located in the corresponding interrupt procedure starting at position $103.

If this doesn't apply to any other interrupts, the driver hasn't been loaded yet and isn't resident. In this case, the function value of CMFInitDriver is FALSE. Otherwise, a driver Reset is activated and the function value TRUE is returned.

Function CMFGetVersion

This function executes driver function 0. The driver's version number is returned as the function value of CMFGetVersion.

Procedure CMFSetStatusByte

This procedure uses driver function 1 to specify that the driver status must always be assigned to the variable CMFStatusByte.

Function CMFSetInstruments

This function checks a CMF file to determine where its instrument data are located and how many instruments are included in the file. Then the instruments are set. You must specify only the pointer identifying the CMF file; the function does the rest.

Depending on whether the instruments were set successfully, the function value is TRUE or FALSE.

Function CMFSetSingleInstruments

This function uses the same driver function to set the instruments. However, you must specify the memory area containing the instrument data, as well as the number of instruments, as function parameters.

This function is useful when you want to use instrument data that isn't stored in a CMF file. The function value is TRUE or FALSE, depending on whether the function is successful.

Procedure CMFSetSysClock

This procedure is used to set the standard system clock rate, which is reset after a song has been completely played back. You don't need to perform the frequency calculation we described in the

section on driver functions. Simply enter the desired rate as a frequency in Hertz.

Procedure CMFSetDriverClock

This function allows you to set the driver clock rate that's used during the playback of a song. Simply specify the desired frequency in Hertz.

Procedure CMFSetTransposeOfs

With this procedure, you can specify the number of half-tone steps each note should be raised or lowered during playback.

This value is entered as either a positive or negative number, depending on whether you want to raise or lower the song's pitch.

Function CMFPlaySong

This function performs the tasks needed to play back a CMF song.

The only parameter you must pass to the function is a pointer identifying the CMF data. The rest is handled by CMFPlaySong.

First the driver clock rate is set to the value specified in the CMF file. Then the instrument definition data is passed to the FM registers.

After this, driver function 6 is called, the music playback begins, and CMFPlaySong returns the value TRUE.

If this doesn't occur, CMFPlaySong returns the value FALSE and CMFErrStat is set to 500.

Function CMFPause/ContinueSong

These two functions allow you to pause a song that's currently being played back and then continue the playback later.

Depending on the function that's called, the global variable CMFSongPaused is set to either TRUE or FALSE.

The function value for both functions is always true if they're executed successfully; otherwise it's false.

Function CMFStopSong

This function aborts the playback of a CMF song. If music data wasn't being played at the time of the function call, the function value is FALSE. Otherwise the returned value is TRUE.

Function CMFResetDriver

This function is used to set the FM chip registers back to their base values. If a song is currently being played, this function cannot be executed. So the function returns the value FALSE.

If the initialization was successful, the value TRUE is returned. You should always call this function at the end of your programs.

With these functions, you can easily play CMF files in the background, while your program performs other tasks in the foreground. This is particularly effective for applications such as title screens.

```
UNIT CMFTool;
{
 ******************************************************************
 *     Unit for Sound Blaster card control in Turbo Pascal 6.0    *
 *                  using the SBFMDRV.COM driver.                 *
 ******************************************************************
 *                        (C) 1992 Abacus                         *
 *                      Author : Axel Stolz                       *
 ******************************************************************
}

INTERFACE

USES Dos;

TYPE
  CMFFileTyp = FILE;
  CMFDataTyp = Pointer;
  CMFHeader = RECORD  { CMF file header structure }
    CMFFileID          : ARRAY[0..3] OF CHAR;  { CMF file ID = 'CTMF'     }
    CMFVersion         : WORD;                 { Version number           }
    CMFInstrBlockOfs   : WORD;                 { Instrument offset        }
    CMFMusicBlockOfs   : WORD;                 { Music data offset        }
    CMFTickPerBeat     : WORD;                 { "Ticks" per beat         }
    CMFClockTicksPS    : WORD;                 { Timer clock rate         }
    CMFFileTitleOfs    : WORD;                 { Music title offset       }
    CMFComposerOfs     : WORD;                 { Composer offset          }
    CMFMusicRemarkOfs  : WORD;                 { Remarks offset           }
    CMFChannelsUsed    : ARRAY[0..15] OF CHAR; { Number of channels used }
    CMFInstrNumber     : WORD;                 { Number of instruments    }
```

```
    CMFBasicTempo     : WORD;                  { Basic music tempo       }
  END;

CONST
   CMFToolVersion      = 'v1.0';

VAR
   CMFStatusByte      : BYTE;       { CMF status variable status   }
   CMFErrStat         : WORD;       { CMF error number variable    }
   CMFDriverInstalled : BOOLEAN;    { Flag -> Driver installed?    }
   CMFDriverIRQ       : WORD;       { Number of IRQs used          }
   CMFSongPaused      : BOOLEAN;    { Flag -> Song paused?         }
   OldExitProc        : Pointer;    { Pointer to old ExitProc      }

PROCEDURE PrintCMFErrMessage;
FUNCTION  CMFGetSongBuffer(VAR CMFBuffer : Pointer; CMFFile :
STRING):BOOLEAN;
FUNCTION  CMFFreeSongBuffer (VAR CMFBuffer : Pointer):BOOLEAN;
FUNCTION  CMFInitDriver : BOOLEAN;
FUNCTION  CMFGetVersion : WORD;
PROCEDURE CMFSetStatusByte;
FUNCTION  CMFSetInstruments(VAR CMFBuffer : Pointer):BOOLEAN;
FUNCTION  CMFSetSingleInstruments(VAR CMFInstrument:Pointer;
No:WORD):BOOLEAN;
PROCEDURE CMFSetSysClock(Frequency : WORD);
PROCEDURE CMFSetDriverClock(Frequency : WORD);
PROCEDURE CMFSetTransposeOfs (Offset : INTEGER);
FUNCTION  CMFPlaySong(VAR CMFBuffer : Pointer) : BOOLEAN;
FUNCTION  CMFStopSong : BOOLEAN;
FUNCTION  CMFResetDriver:BOOLEAN;
FUNCTION  CMFPauseSong : BOOLEAN;
FUNCTION  CMFContinueSong : BOOLEAN;

IMPLEMENTATION

TYPE
   TypeCastTyp = ARRAY [0..6000] of Char;

VAR
   Regs : Registers;
   CMFIntern : ^CMFHeader; { Internal pointer to CMF structure }

PROCEDURE PrintCMFErrMessage;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Displays SB error on the screen as text, without changing
 *           error status.
}
BEGIN
   CASE CMFErrStat OF
      100 : Write(' SBFMDRV sound driver not found ');
      110 : Write(' Driver reset unsuccessful ');
```

```
      200 : Write(' CMF file not found ');
      210 : Write(' No memory free for CMF file ');
      220 : Write(' File not in CMF format ');

      300 : Write(' Memory allocation error occurred ');

      400 : Write(' Too many instruments defined ');

      500 : Write(' CMF data could not be played ');
      510 : Write(' CMF data could not be stopped ');
      520 : Write(' CMF data could not be paused ');
      530 : Write(' CMF data could not be continued ');
      END;
   END;

FUNCTION Exists (Filename : STRING):BOOLEAN;
{
 * INPUT   : File name as string
 * OUTPUT  : TRUE if file is available, FALSE if not
 * PURPOSE : Checks for the existence of a file, and returns an
             appropriate Boolean expression.
}
VAR
   F : File;
BEGIN
   Assign(F,Filename);
{$I-}
   Reset(F);
   Close(F);
{$I+}
   Exists := (IoResult = 0) AND (Filename <> '');
   END;

PROCEDURE AllocateMem (VAR Pt : Pointer; Size : LongInt);
{
 * INPUT   : Pointer variable as pointer, buffer size as LongInt
 * OUTPUT  : Buffer pointer in variable or NIL
 * PURPOSE : Reserves as many bytes as Size allows, then sets the
             pointer in the Pt variable. If not enough memory is
             available, Pt is set to NIL.
}
VAR
   SizeIntern : WORD;     { Pointer size for internal calculation }
BEGIN
   Inc(Size,15);                  { Increment buffer by 15 ...    }
   SizeIntern := (Size shr 4);    { and divide it by 16           }
   Regs.AH := $48;                { Place DOS function $48 in AH }
   Regs.BX := SizeIntern;         { Place internal size in BX    }
   MsDos(Regs);                   { Reserve memory               }
   IF (Regs.BX <> SizeIntern) THEN Pt := NIL
   ELSE Pt := Ptr(Regs.AX,0);
   END;

FUNCTION  CheckFreeMem (VAR CMFBuffer : Pointer; CMFSize : LongInt):BOOLEAN;
```

```
{
 * INPUT   : Buffer variable as pointer, desired size as LongInt
 * OUTPUT  : Pointer to buffer, TRUE/FALSE, as in AllocateMem
 * PURPOSE : Ensures that enough memory has been allocated for CMF file.
}
BEGIN
   AllocateMem(CMFBuffer,CMFSize);
   CheckFreeMem := CMFBuffer <> NIL;
   END;

FUNCTION  CMFGetSongBuffer(VAR CMFBuffer : Pointer; CMFFile :
STRING):BOOLEAN;
{
 * INPUT   : Buffer variable as pointer, file name as string
 * OUTPUT  : Pointer to buffer for CMF data, TRUE/FALSE
 * PURPOSE : Loads file into memory; returns TRUE if load was successful,
             FALSE if not.
}
CONST
   FileCheck : STRING[4] = 'CTMF';
VAR
   CMFFileSize : LongInt;
   FPresent    : BOOLEAN;
   VFile       : CMFFileTyp;
   Segs        : WORD;
   Read        : WORD;
   Checkcount  : BYTE;

BEGIN
   FPresent := Exists(CMFFile);

{ CMF file could not be found }
   IF Not(FPresent) THEN BEGIN
      CMFGetSongBuffer := FALSE;
      CMFErrStat   := 200;
      EXIT
      END;

   Assign(VFile,CMFFile);
   Reset(VFile,1);
   CMFFileSize := Filesize(VFile);
   AllocateMem(CMFBuffer,CMFFileSize);

{ Insufficient memory for CMF file }
   IF (CMFBuffer = NIL) THEN BEGIN
      Close(VFile);
      CMFGetSongBuffer := FALSE;
      CMFErrStat   := 210;
      EXIT;
      END;

   Segs := 0;
   REPEAT
```

```
Blockread(VFile,Ptr(seg(CMFBuffer^)+4096*Segs,Ofs(CMFBuffer^))^,$FFFF,Read);
      Inc(Segs);
      UNTIL Read = 0;
   Close(VFile);

{ File not in CMF format }
   CMFIntern := CMFBuffer;
   CheckCount := 1;
   REPEAT
      IF FileCheck[CheckCount] = CMFIntern^.CMFFileID[CheckCount-1]
         THEN Inc(CheckCount)
         ELSE CheckCount := $FF;
      UNTIL CheckCount >= 3;
   IF NOT(CheckCount = 3) THEN BEGIN
      CMFGetSongBuffer := FALSE;
      CMFErrStat   := 220;
      EXIT;
      END;

{ Load was successful }
   CMFGetSongBuffer := TRUE;
   CMFErrStat   := 0;
   END;

FUNCTION CMFFreeSongBuffer (VAR CMFBuffer : Pointer):BOOLEAN;
{
 * INPUT   : Pointer to buffer
 * OUTPUT  : None
 * PURPOSE : Frees memory allocated for CMF file.
}
BEGIN
   Regs.AH := $49;              { Place DOS function $49 in AH }
   Regs.ES := seg(CMFBuffer^);  { Place memory segment in ES   }
   MsDos(Regs);                 { Free memory                  }
   CMFFreeSongBuffer := TRUE;
   IF (Regs.AX = 7) OR (Regs.AX = 9) THEN BEGIN
      CMFFreeSongBuffer := FALSE;
      CMFErrStat := 300         { DOS error on release         }
      END;
   END;


FUNCTION CMFInitDriver : BOOLEAN;
{
 * INPUT   : None
 * OUTPUT  : TRUE if driver is found and installed, FALSE if not
 * PURPOSE : Checks for SBFMDRV.COM resident in memory, and resets
             the driver
}

CONST
   DriverCheck :STRING[5] = 'FMDRV'; { String to search for in SBFMDRV }
VAR
```

```
   ScanIRQ,
   CheckCount  : BYTE;
   IRQPtr,
   DummyPtr    : Pointer;

BEGIN
{ Possible SBFMDRV interrupts lie in range $80 - $BF }
   FOR ScanIRQ := $80 TO $BF DO BEGIN
      GetIntVec(ScanIRQ, IRQPtr);
      DummyPtr := Ptr(Seg(IRQPtr^), $102);

{ Check for string 'FMDRV' in interrupt program. }
{ If so, then SBFMDRV can be managed.            }
      CheckCount := 1;
      REPEAT
         IF DriverCheck[CheckCount] = TypeCastTyp(DummyPtr^)[CheckCount]
            THEN Inc(CheckCount)
            ELSE CheckCount := $FF;
         UNTIL CheckCount >= 5;

      IF (CheckCount = 5) THEN BEGIN
{ String found; reset executed }
         Regs.BX := 08;
         CMFDriverIRQ := ScanIRQ;
         Intr(CMFDriverIRQ, Regs);
         IF Regs.AX = 0 THEN
            CMFInitDriver := TRUE
         ELSE BEGIN
            CMFInitDriver := FALSE;
            CMFErrStat    := 110;
            END;
         Exit;
         END
      ELSE BEGIN
{ String not found }
         CMFInitDriver := FALSE;
         CMFErrStat := 100;
         END;
      END;
   END;

FUNCTION CMFGetVersion : WORD;
{
 * INPUT   : None
 * OUTPUT  : Version number in high byte, subversion number in low byte
 * PURPOSE : Gets version number from SBFMDRV driver.
}

BEGIN
   Regs.BX := 0;
   Intr(CMFDriverIRQ,Regs);
   CMFGetVersion := Regs.AX;
   END;
```

```
PROCEDURE CMFSetStatusByte;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Place driver status byte in CMFStatusByte variable.
}

BEGIN
   Regs.BX:= 1;
   Regs.DX:= Seg(CMFStatusByte);
   Regs.AX:= Ofs(CMFStatusByte);
   Intr(CMFDriverIRQ, Regs);
   END;

FUNCTION CMFSetInstruments(VAR CMFBuffer : Pointer):BOOLEAN;
{
 * INPUT   : CMF file buffer as pointer
 * OUTPUT  : TRUE or FALSE; depends on success in setting instruments
 * PURPOSE : Sets SB card FM registers to instrumentation as stated
             in the loaded CMF file.
}

BEGIN
    CMFIntern := CMFBuffer;
    IF CMFIntern^.CMFInstrNumber > 128 THEN BEGIN
       CMFErrStat := 400;
       CMFSetInstruments := FALSE;
       Exit;
       END;
    Regs.BX := 02;
    Regs.CX := CMFIntern^.CMFInstrNumber;
    Regs.DX := Seg(CMFBuffer^);
    Regs.AX := Ofs(CMFBuffer^)+CMFIntern^.CMFInstrBlockOfs;
    Intr(CMFDriverIRQ, Regs);
    CMFSetInstruments := TRUE;
   END;

FUNCTION CMFSetSingleInstruments(VAR CMFInstrument:Pointer; No:WORD):BOOLEAN;
{
 * INPUT   : CMF instrument data pointer, number of instruments as WORD
 * OUTPUT  : TRUE or FALSE; depends on success in setting instruments
 * PURPOSE : Sets SB FM registers to instrument values corresponding to the
             data structure following the CMFInstrument pointer.
}

BEGIN
    IF No > 128 THEN BEGIN
       CMFErrStat := 400;
       CMFSetSingleInstruments := FALSE;
       Exit;
       END;
    Regs.BX := 02;
    Regs.CX := No;
```

```
    Regs.DX := Seg(CMFInstrument^);
    Regs.AX := Ofs(CMFInstrument^);
    Intr(CMFDriverIRQ, Regs);
    CMFSetSingleInstruments := TRUE;
   END;

PROCEDURE CMFSetSysClock(Frequency : WORD);
{
 * INPUT   : System timer clock rate as WORD
 * OUTPUT  : None
 * PURPOSE : Sets default value of timer 0 to new value.
}

BEGIN
   Regs.BX := 03;
   Regs.AX := (1193180 DIV Frequency);
   Intr(CMFDriverIRQ, Regs);
   END;

PROCEDURE CMFSetDriverClock(Frequency : WORD);
{
 * INPUT   : Timer clock rate as WORD
 * OUTPUT  : None
 * PURPOSE : Sets driver timer frequency to new value.
}

BEGIN
   Regs.BX := 04;
   Regs.AX := (1193180 DIV Frequency);
   Intr(CMFDriverIRQ, Regs);
   END;

PROCEDURE CMFSetTransposeOfs (Offset : INTEGER);
{
 * INPUT   : Offset as WORD. Value specifies the amount of note
             transposition in half steps.
 * OUTPUT  : None
 * PURPOSE : Transposes all notes in the CMF file by "Offset."
}

BEGIN
   Regs.BX := 05;
   Regs.AX := Offset;
   Intr(CMFDriverIRQ, Regs);
   END;

FUNCTION CMFPlaySong(VAR CMFBuffer : Pointer) : BOOLEAN;
{
 * INPUT   : Pointer to song data
 * OUTPUT  : TRUE if start is successful, FALSE if not
 * PURPOSE : Initializes all important parameters and starts song playback.
}

VAR
```

```
   Check : BOOLEAN;
BEGIN
   CMFIntern := CMFBuffer;
{ Set driver clock frequency }
   CMFSetDriverClock(CMFIntern^.CMFClockTicksPS);
{ Set instruments }
   Check := CMFSetInstruments(CMFBuffer);
   IF Not(Check) THEN Exit;
   Regs.BX := 06;
   Regs.DX := Seg(CMFIntern^);
   Regs.AX := Ofs(CMFIntern^)+CMFIntern^.CMFMusicBlockOfs;
   Intr(CMFDriverIRQ, Regs);

   IF Regs.AX = 0 THEN BEGIN
      CMFPlaySong := TRUE;
      CMFSongPaused := FALSE;
      END
   ELSE BEGIN
      CMFPlaySong := FALSE;
      CMFErrStat := 500;
      END;
   END;

FUNCTION CMFStopSong : BOOLEAN;
{
 * INPUT   : None
 * OUTPUT  : TRUE or FALSE; depends on success of stop
 * PURPOSE : Attempts to stop song playback.
}

BEGIN
   Regs.BX := 07;
   Intr(CMFDriverIRQ, Regs);
   IF Regs.AX = 0 THEN
      CMFStopSong := TRUE
   ELSE BEGIN
      CMFStopSong := FALSE;
      CMFErrStat  := 510;
      END;
   END;

FUNCTION CMFResetDriver:BOOLEAN;
{
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Resets driver to starting status.
}

BEGIN
   Regs.BX := 08;
   Intr(CMFDriverIRQ, Regs);
   IF Regs.AX = 0 THEN
      CMFResetDriver := TRUE
   ELSE BEGIN
```

```
      CMFResetDriver := FALSE;
      CMFErrStat    := 110;
      END;
   END;

FUNCTION CMFPauseSong : BOOLEAN;
{
 * INPUT   : None
 * OUTPUT  : TRUE or FALSE; depends on success of pause
 * PURPOSE : Attempts to pause song playback. If pause is possible, this
             function sets the CMFSongPaused variable to TRUE.
}

BEGIN
   Regs.BX := 09;
   Intr(CMFDriverIRQ, Regs);
   IF Regs.AX = 0 THEN BEGIN
      CMFPauseSong  := TRUE;
      CMFSongPaused := TRUE;
      END
   ELSE BEGIN
      CMFPauseSong := FALSE;
      CMFErrStat   := 520;
      END;
   END;

FUNCTION CMFContinueSong : BOOLEAN;
{
 * INPUT   : None
 * OUTPUT  : TRUE or FALSE; depends on success of continuation
 * PURPOSE : Attempts to continue playback of a paused song. If continuation
             is possible, this function sets the CMFSongPaused variable to
             FALSE.
}

BEGIN
   Regs.BX := 10;
   Intr(CMFDriverIRQ, Regs);
   IF Regs.AX = 0 THEN BEGIN
      CMFContinueSong  := TRUE;
      CMFSongPaused    := FALSE;
      END
   ELSE BEGIN
      CMFContinueSong := FALSE;
      CMFErrStat      := 530;
      END;
   END;

{$F+}
PROCEDURE CMFToolsExitProc;
{$F-}
{
 * INPUT   : None
 * OUTPUT  : None
```

```
 * PURPOSE : Resets the status byte address, allowing this program to exit.
}
BEGIN
   Regs.BX:= 1;
   Regs.DX:= 0;
   Regs.AX:= 0;
   Intr(CMFDriverIRQ, Regs);
   ExitProc := OldExitProc;
   END;


BEGIN
{ Reset old ExitProc to the Tool unit proc }
   OldExitProc := ExitProc;
   ExitProc := @CMFToolsExitProc;
{ Initialize variables }
   CMFErrStat := 0;
   CMFSongPaused := FALSE;
{ Initialize driver }
   CMFDriverInstalled := CMFInitDriver;
   IF CMFDriverInstalled THEN BEGIN
      CMFStatusByte := 0;
      CMFSetStatusByte;
      END;
   END
```

CMFDEMO

We'll use a program located in the CMFDEMO.PAS file to explain how to use the functions in CMFTOOL.TPU.

Set memory limit

First the {$M} compiler directive must be used to specify the program's upper memory limit. In this example, the limit consists of 16K for the stack and 0 to 64K for the program.

This should provide a sufficient amount of available memory for the CMF data. Our discussion of the memory limits for the VOCTOOL.PAS unit also applies in this situation.

The Uses statement is then used to automatically add and initialize the CMFTOOL unit.

TextNumError procedure

The TextNumError procedure displays the appropriate error number and message when an error occurs during the program. Then the program is interrupted using HALT, and the error level is passed to DOS, indicating which error number caused the interruption.

In your own programs, you should insert an error processing procedure instead of terminating the program.

Running CMFDEMO

Make sure that the STARFM.CMF file is in the same directory as CMFDEMO.EXE. Run SBFMDRV.COM, then run CMFDEMO.EXE.

Main program

The main program first checks whether the desired driver has been initialized successfully. If it hasn't, the appropriate error number is displayed and the program is ended. The variable CMFDriverInstalled is checked to determine whether this procedure was successful.

If CMFDriverInstalled contains the value TRUE, the program is continued. Then the command line parameters are checked.

If parameters haven't been specified, the STARFM.CMF file is used as the default CMF file that must be played. This is one of the sample files that's included with the Sound Blaster card.

However, if a parameter has been specified, it's assigned to the SongName variable.

Next the driver version number of SBFMDRV.COM and the interrupt that's being used are displayed.

The driver version number that's returned by the driver function 0 may differ from the version number that's displayed when SBFMDRV.COM is started. However, this doesn't mean that function 0 has been used incorrectly.

This indicates there is a difference between the internal version number and the number that's displayed.

Then the specified CMF file is loaded. The variable SongBuffer is used as a pointer variable, which points to the start of the CMF data in memory once the file has been loaded.

After this, the song may be transposed. The half-tone value used in this example is 0, so the song's pitch isn't changed. However, you can experiment with this value.

Then the playback of the song is started. Throughout the duration of the playback, the status byte is available through the variable CMFStatusByte.

The program runs until either a key is pressed or until the end of the music data has been reached.

When a keystroke is detected, the song is interrupted. This is unnecessary once the end of the song has been reached. The driver is then initialized again, and the memory containing the CMF data is freed.

All these steps are needed to play back a song in the background of a program.

```
Program CMFDemo;
{
 *************************************************************************
 *  Demonstration program for CMFTOOL unit, written in Turbo Pascal 6.0  *
 *************************************************************************
 *                          (C) 1992 Abacus                              *
 *                        Author : Axel Stolz                            *
 *************************************************************************
 * Memory limits must be allocated by the program, or the program will   *
 * use all available memory, leaving no additional memory for the driver *
 * and sample data. The memory parameter must be passed to the           *
 * main program.                                                         *
 *************************************************************************
}


{$M 16384,0,65535}

Uses CMFTool,Crt;

VAR
   Check      : BOOLEAN;    { Boolean variable for different checks }
   SongName   : String;     { String for CMF file name              }
   SongBuffer : CMFDataTyp; { Data buffer for CMF file              }

PROCEDURE TextNumError;
{
 * INPUT   : None; data comes from CMFErrStat global variable
 * OUTPUT  : None
 * PURPOSE : Displays SB error on the screen as text, including the
             error number. Program then ends at the error level
             corresponding to the error number.
}
BEGIN
   Write(' Error #',CMFErrStat:3,': ');
   PrintCMFErrMessage;
   WriteLn;
   Halt(CMFErrStat);
   END;

BEGIN
   ClrScr;
{ Displays error if SBFMDRV driver has not been installed }
   IF Not (CMFDriverInstalled) THEN TextNumError;
```

```
{
 If no song name is included with command line parameters,
 program searches for the default name (here STARFM.CMF).
}
   IF ParamCount = 0 THEN SongName := 'STARFM.CMF'
                     ELSE SongName := ParamStr(1);

{ Display driver's version and subversion numbers }
   GotoXY(28,5);
   Write  ('SBFMDRV Version ',Hi(CMFGetVersion):2,'.');
   WriteLn(Lo(CMFGetVersion):2,' loaded');

{ Display interrupt number in use }
   GotoXY(24,10);
   Write  ('System interrupt (IRQ) ');
   WriteLn(CMFDriverIRQ:3,' in use');
   GotoXY(35,15);
   WriteLn('Song Status');
   GotoXY(31,23);
   WriteLn('Song name: ',SongName);

{ Load song file }
   Check := CMFGetSongBuffer(SongBuffer,SongName);
   IF NOT(Check) THEN TextNumError;

{
 CMFSetTransposeOfs() controls transposition down or up of the loaded song
 (positive values transpose up, negative values transpose down). The value
 0 plays the loaded song in its original key.
}
   CMFSetTransposeOfs(0); { Experiment with this value }

{ Play song }
   Check := CMFPlaySong(SongBuffer);
   IF NOT(Check) THEN TextNumError;

{ During playback, display status byte }
   REPEAT
      GotoXY(41,17);Write(CMFStatusByte:3);
      UNTIL (KeyPressed OR (CMFStatusByte = 0));

{ Stop playback if user presses a key }
   IF KeyPressed THEN BEGIN
      Check := CMFStopSong;
      IF NOT(Check) THEN TextNumError;
      END;

{ Re-initialize driver }
   Check := CMFResetDriver;
   IF NOT(Check) THEN TextNumError;

{ Free song file memory }
   Check := CMFFreeSongBuffer(SongBuffer);
   IF NOT(Check) THEN TextNumError;
```

```
END.
```

Many music files are available in CMF format. Other music files, such as ROL files for the AdLib card, can only be used after they are converted to CMF format with the ROL2CMF program.

ROL2CMF is a program that you can find on many bulletin boards. This helps you increase your supply of music files.

5.3.6 CMF programming using Borland C++ 3.1

In this section we'll present a module written with Borland C++ 3.1. This module corresponds to the preceding Turbo Pascal unit and uses the SBFMDRV.COM driver. This module is located on the companion diskette.

Similar to the digital channel, the interface to this driver is stored in a file using the .H filename extension. The complete filename is CMFTOOL.H.

It should be easy to transfer this program to another C compiler because an internal assembler isn't needed.

The module contains all driver functions, except for function 11, which is the user-defined function.

Since the structure of this module is based on the VOCTOOL.H module, you can work with and compare the two modules easily.

First the #define statement is used to define several constants for the compilation. This requires, among other things, the offset values within the CMF file that are needed to read the necessary playback data.

The data type definition CMFPTR is important. It can be used in your programs to indicate that a particular variable is a pointer to a set of CMF data.

Then the header of a CMF file is provided as a data type. This allows you to access the header's individual values within your own programs through the cmf_header structure.

The version number of the CMFTOOL module is assigned to the constant cmftool_version as a character string. The current version number is V1.0. Remember to update this number each time you modify the module.

Next is the definition of the module's global variables, which we'll describe separately later.

From the complete list of prototypes you'll be able to see that the module contains a total of 17 functions.

Global error messages

Since the first function displays the global error numbers as text, it's called _print_cmf_errmessage().

The numbers used for the module's global errors are identical to those used in the Pascal unit.

Each time an error occurs with the use of the CMFTOOL.H module, the global error variable cmf_err_stat is set to the appropriate error number, which can then be read from the variable.

This allows your program to react to each error individually. However, you must remember to have the program reset the error variable to zero once you've processed the error. Otherwise the variable will retain the current error number until the next error occurs.

CMFTOOL error numbers

Error numbers 1xx are reserved for errors that may occur during the initialization of the driver.

Error 100

Error 100 indicates that the SBFMDRV.COM driver is not resident.

Error 110

Error 110 is used when the _cmf_reset() function call was unsuccessful.

2xx error numbers are used for errors that may occur when the CMF files are loaded.

Error 200

Error 200 indicates that the specified CMF file couldn't be found. Either an incorrect name was specified or the file wasn't located at the indicated path.

Error 210

Error 210 is activated when there isn't enough free memory available to load the specified CMF file.

In this case you've either actually used all your memory or the DOS function _dos_allocmem() cannot find a free memory block.

However, unlike VOC files, CMF files are usually small enough so they should easily fit into the available memory.

Error 220 Error 220 indicates that the loaded file isn't a CMF file. For this identity check, the file header's first four bytes are checked. These bytes must contain the characters "CTMF".

Error 300 Error 300 is used when DOS cannot free a reserved memory area.

However, since C already contains the _dos_freemem() function, it isn't actually necessary to create a separate function, such as CMFFreeBuffer, from the Pascal version. Therefore, the error number 300 will never be triggered within this module.

The error number is included in this module for the same reasons as in the VOCTOOL.H module—so it cannot be used for other purposes. This would destroy the identical structure of this module and the Pascal unit.

If you want to use error number 300 in this C module, you must write a function that sets the error number 300 whenever the _dos_freemem() function call isn't executed successfully.

Error 400 Error 400 indicates that you've tried to assign more than 128 instrument definitions to the driver.

5xx error numbers occur with the playback of CMF data.

Error 500 Error 500 occurs when a CMF song couldn't be started. Usually this is because another song is currently being played back.

Error 510 Error number 510 is used to indicate that a song playback couldn't be interrupted. This is because a playback wasn't in progress at the time of the function call.

Error 520 Error number 520 indicates that a music playback couldn't be paused because a playback wasn't in progress at the time of the function call.

Error 530 Error 530 always occurs when a paused music playback couldn't be continued.

CMFTOOL functions

In this section we'll describe the functions of the CMFTOOL.H module.

_print_cmf_errmessage()

This function displays a CMF error on your screen as text. You can easily incorporate this function into your own error processing

procedure because it uses only a printf() statement with no line feed.

This function is useful as a reminder of the module's error numbers when you're programming with the module.

_cmf_get_songbuffer()

This function performs all the tasks needed to load a CMF file into memory. The only parameter you must pass to this function is the name of the file that must be loaded in the form of a string variable.

The function then tries to open the file and reserve the required memory area. Then the data is loaded into memory.

If the file was loaded successfully, the function returns the address of the data in memory.

If the file couldn't be loaded, the return value is zero. In this case, you must check cmf_err_stat to determine which error has occurred.

_cmf_init_driver()

This function searches for the installed driver SBFMDRV. It examines interrupts 0x80 through 0xBF to determine whether the characters "FMDRV" are located in the corresponding interrupt procedure, starting at position 0x103.

If this character sequence isn't found behind these interrupts, then the driver isn't resident. In this case, the function value of _cmf_init_driver() is FALSE. Otherwise, a driver Reset is performed and the function value TRUE is returned.

_cmf_getversion()

The function executes the driver function 0. The function value contains the version number in the form of a WORD.

_cmf_set_status_byte()

This function uses driver function 1 to specify that the current driver status must always be written to the global variable cmf_status_byte.

_cmf_set_instruments()

This function checks a specific CMF file for the location of its instrument data, determines the number of instruments, and then sets the instruments. You must specify only the pointer identifying the loaded CMF file; the function handles the rest.

Depending on whether the instruments were set successfully, the returned function value is either TRUE or FALSE.

_cmf_set_singleinstrument()

This function uses the same driver function for setting the instruments. However, you must specify the address for the instrument data, as well as the number of instruments that must be set manually.

Use this function when you want to use instrument data that isn't located in a given CMF file. Depending on the success of the function call, the returned function value is either TRUE or FALSE.

_cmf_set_sysclock()

This function is used to set the standard system clock rate that must be reset once a music playback has been completed.

You don't have to perform the calculation that was needed for the driver function. Simply enter the desired rate as a frequency of the unit Hertz.

_cmf_set_driverclock()

With this function you can set the driver clock rate, which is responsible for controlling the speed of a music playback. Again, you must specify only the desired frequency in Hertz.

_cmf_set_transposeofs()

This function is used to specify the number of half-tone steps each note must be raised or lowered during a playback.

The number of half-tone steps is entered as either a positive or negative value, depending on whether you want to transpose the song up or down.

_cmf_play_song()

This function performs all the tasks required for playing back a CMF song.

The only parameter that must be specified is a pointer identifying the start of the loaded CMF data. The rest is handled by _cmf_play_song().

First the driver clock rate is set to the value specified in the CMF file. Then the instrument definitions are passed to the FM registers.

After this, driver function 6 is called, the music playback is started, and _cmf_play_song() receives the value TRUE.

If the driver doesn't start the playback, the function value is FALSE and cmf_err_stat is set to 500.

_cmf_pause/continue_song()

These two functions allow you to pause a song that's being played back and to continue the playback later. Depending on which function is called, the global variable cmf_song_paused is set to either TRUE or FALSE.

For either function, the function value is TRUE if the function was successful and FALSE if it wasn't.

_cmf_stop_song()

This function is used to terminate the playback of a CMF song. If a music playback wasn't in progress at the time of the function call, the function value is FALSE. When the function call is successful, the value TRUE is returned.

_cmf_reset_driver()

This function is used to set the FM chip registers back to their default values. This function cannot be executed if a song is currently being played back; in this case the function value FALSE is returned.

If the initialization is successful, the value TRUE is returned. You should always call this function at the end of your programs.

_cmf_prepare_use()

Since the C module MODTOOL.H doesn't perform an automatic initialization of all relevant values for the module, which is done in the Pascal version, the _cmf_prepare_use() function has been added.

This function uses _cmf_init_driver() to check whether the driver has been found. In this case, the address of the variable cmf_status_byte is passed to the driver. You can determine the driver's status through this variable. Then the function is ended with the value TRUE.

However, if the driver hasn't been located yet, the value FALSE is returned and the function is ended.

You should place this function at the start of your own programs to obtain the necessary information about the driver.

_cmf_terminate_use()

This function is needed to prevent the memory resident driver SBFMDRV from continuing to write status values to the address of the variable cmf_status_byte, even though you've already ended your program and a new program may be located at this address. Obviously this causes some undesirable effects.

So you should always call the _cmf_terminate_use() function at the end of your programs.

With these functions, you can easily play back CMF files in the background, while your program performs other tasks in the foreground.

This is particularly effective for title screens. However, you'll find many other ways to use background music.

```
/*
 *********************************************************************
 * Module for controlling the Sound Blaster card in Borland C++ 3.1 *
 *                using the SBFMDRV.COM driver.                      *
 *********************************************************************
 *                      (C) 1992 Abacus                              *
 *                    Author : Axel Stolz                            *
 *********************************************************************
*/

#include <io.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <dos.h>
#include <bios.h>
#include <fcntl.h>

#define  FALSE  0
#define  TRUE   1

#define  INSOFF 6  /* Memory offset Instr_Block_Off    */
#define  MUSOFF 8  /* Memory offset Music_Block_Off    */
#define  CTPS   12 /* Memory offset Clock_Ticks_PS     */
#define  INSNUM 36 /* Memory offset Instrument_Number  */

#define  DWORD  unsigned long
#define  WORD   unsigned int
#define  BYTE   unsigned char
#define  CMFPTR char far*

typedef struct {
   char   cmf_fileid[4];         /* CMF file ID = 'CTMF'     */
   WORD   cmf_version;           /* Version number           */
   WORD   cmf_inst_block_ofs;    /* Offset for instruments   */
   WORD   cmf_music_block_ofs;   /* Offset for music data    */
   WORD   cmf_ticks_per_beat;    /* "Ticks" per beat         */
   WORD   cmf_clockticks_ps;     /* Timer clock rate         */
   WORD   cmf_file_title_ofs;    /* Offset music title       */
   WORD   cmf_composer_ofs;      /* Offset music composer    */
   WORD   cmf_music_remark_ofs;  /* Offset music remarks     */
   char   cmf_channel_used[16];  /* Number of channels used  */
   WORD   cmf_instr_number;      /* Number of instruments    */
   WORD   cmf_basic_tempo;       /* Basic tempo              */
   } cmf_header;

   const char cmftool_version[] = "v1.0";

   BYTE   cmf_status_byte;       /* Variable for CMF status       */
   WORD   cmf_err_stat;          /* Variable for CMF error number */
   BYTE   cmf_driverinstalled;   /* Flag for installed driver     */
   WORD   cmf_driverirq;         /* Number of IRQs used           */
   BYTE   cmf_songpaused;        /* Flag for paused song          */

/*
 ********************************************
 * Prototypes for created functions         *
 ********************************************
 */

    void    _print_cmf_errmessage(void);
    CMFPTR  _cmf_get_songbuffer(char *cmffile);
    BYTE    _cmf_init_driver(void);
    WORD    _cmf_getversion(void);
    void    _cmf_set_status_byte(void);
    BYTE    _cmf_set_instruments(CMFPTR cmf_buffer);
    BYTE    _cmf_set_singleinstruments(CMFPTR cmf_instrument, WORD No);
```

```
    void     _cmf_set_sysclock(WORD frequency);
    void     _cmf_set_driverclock(WORD frequency);
    void     _cmf_set_transposeofs(int offset);
    BYTE     _cmf_play_song(CMFPTR cmf_buffer);
    BYTE     _cmf_stop_song(void);
    BYTE     _cmf_reset_driver(void);
    BYTE     _cmf_pause_song(void);
    BYTE     _cmf_continue_song(void);
    BYTE     _cmf_prepare_use(void);
    void     _cmf_terminate_use(void);

void _print_cmf_errmessage(void)
/*
 * INPUT   : none
 * OUTPUT  : none
 * PURPOSE : Displays SB errors on screen as text, without
 *           changing the error status.
*/
{
   switch (cmf_err_stat) {
    case 100 : printf(" SBFMDRV sound driver not loaded");break;
    case 110 : printf(" Driver reset unsuccessful ");break;

    case 200 : printf(" CMF file not found ");break;
    case 210 : printf(" No free memory for CMF file ");break;
    case 220 : printf(" File is not in CMF format ");break;

    case 300 : printf(" Memory allocation error occurred ");break;

    case 400 : printf(" Too many instruments defined ");break;

    case 500 : printf(" CMF data could not be played ");break;
    case 510 : printf(" CMF data could not be stopped ");break;
    case 520 : printf(" CMF data could not be paused ");break;
    case 530 : printf(" CMF data could not be continued ");break;
      };
   };


CMFPTR  _cmf_get_songbuffer(char *cmffile)
/*
 * INPUT   : Variable for buffer as pointer, filename as string
 * OUTPUT  : Pointer to buffer with CMF data, TRUE/FALSE
 * PURPOSE : Loads a file into memory and returns the value TRUE when
             successfully loaded, otherwise FALSE.
 */
{
const char filecheck[4] = "CTMF";

   int        filehandle;
   long       filesize;
   WORD       blocksize;
   WORD       byte_read ;
   char huge *filepointer;
```

```
   CMFPTR     bufferaddress;
   BYTE       check;
   WORD       segment;

/* CMF file could not be found */
   if (_dos_open(cmffile,O_RDONLY,&filehandle) != 0)
      {
       cmf_err_stat = 200;
       return(FALSE);
      }

   filesize = filelength(filehandle);
   blocksize = (filesize + 15L) /16;

/* Insufficient memory for the CMF file */
   check = _dos_allocmem(blocksize,&segment);
   if (check != 0)
      {
       cmf_err_stat = 210;
       return(FALSE);
      }

   FP_SEG(bufferaddress) = segment;
   FP_OFF(bufferaddress) = 0;

   filepointer = (char huge*)bufferaddress;

/* Load the CMF file */
   do
      {
       _dos_read(filehandle,filepointer,0x8000,&byte_read);
       filepointer += byte_read ;
      } while (byte_read == 0x8000);

   _dos_close(filehandle);

/* The file is not in CMF format */
   for (check = 0; check < 4; check++)
      {
       if (filecheck[check] != bufferaddress[0+check])
          {
           check = 0xff;
           cmf_err_stat = 210;
           return(NULL);
          }
      }

/* Loading successful */
   cmf_err_stat = 0;

   return(bufferaddress);
}

BYTE _cmf_init_driver(void)
```

```
/*
 * INPUT   : none
 * OUTPUT  : TRUE, if driver is found and initialized, otherwise FALSE
 * PURPOSE : Checks whether SBFMDRV.COM is resident in memory and then
             resets the driver.
 */
{
const char drivercheck[5] = "FMDRV"; /* String in SBFMDRV being sought */

   union REGS regs;
   BYTE   scan_irq;
   WORD   check;
   char far* irq_ptr;
   char far* dummy;


/* Possible interrupts for SBFMDRV are interrupts from 0x80 to 0xBF */
   for (scan_irq = 0x80; scan_irq < 0xC0; scan_irq++)
      {
       irq_ptr = (char far*)getvect(scan_irq);
       FP_SEG(dummy) = FP_SEG(irq_ptr);
       FP_OFF(dummy) = 0x103;

/* Check interrupt program for FMDRV string. */
/* Only in this case can it be SBFMDRV.      */
       for (check = 0; check < 5; check++)
          if (drivercheck[check] != dummy[check])
            check = 0xFE;

/* String found. Will reset now */
       if (check == 5)
       {
          cmf_driverirq = scan_irq;
          scan_irq = 0xc0;
          regs.x.bx = 8;
          int86(cmf_driverirq,&regs, &regs);

          if (regs.x.ax != 0)
             {
              cmf_err_stat = 110;
              return(FALSE);
             }
       }
       else cmf_err_stat = 100;
      }
  if (cmf_err_stat == 100) return (FALSE);
  else return(TRUE);
}


WORD _cmf_getversion(void)
/*
 * INPUT   : none
 * OUTPUT  : Main version number in high byte, subversion number in low byte
```

```
 * PURPOSE : Reads version number from SBFMDRV driver.
 */

{
   union REGS regs;

   regs.x.bx = 0;
   int86(cmf_driverirq,&regs,&regs);
   return(regs.x.ax);
   };


void _cmf_set_status_byte(void)
/*
 * INPUT   : none
 * OUTPUT  : none
 * PURPOSE : Places driver status value in cmf_status_byte variable.
 */

{
   union REGS regs;

   regs.x.bx = 1;
   regs.x.dx = FP_SEG(&cmf_status_byte);
   regs.x.ax = FP_OFF(&cmf_status_byte);
   int86(cmf_driverirq, &regs, &regs);
   };

BYTE _cmf_set_instruments(CMFPTR cmf_buffer)
/*
 * INPUT   : Buffer of CMF data as pointer
 * OUTPUT  : TRUE/FALSE, depending on whether instrument settings successful
 * PURPOSE : Sets SB card FM registers to instrument values contained
             in the loaded CMF file.
 */

{
    union REGS regs;

    if ((WORD)cmf_buffer[INSNUM] > 128)
       {
        cmf_err_stat = 400;
        return(FALSE);
       }

    regs.x.bx = 2;
    regs.x.cx = (WORD)cmf_buffer[INSNUM];
    regs.x.dx = FP_SEG(cmf_buffer);
    regs.x.ax = FP_OFF(cmf_buffer)+(WORD)cmf_buffer[INSOFF];
    int86(cmf_driverirq, &regs, &regs);
    return(TRUE);
   };

BYTE _cmf_set_singleinstruments(CMFPTR cmf_instrument, WORD No)
```

```
/*
 * INPUT   : Buffer of CMF instrument data as pointer,
             Number of instruments as WORD
 * OUTPUT  : TRUE/FALSE, depending on whether instrument setting successful
 * PURPOSE : Sets the SB card FM registers to instrument values that
             correspond to the data structure hidden behind the
             cmf_instrument pointer.
 */

{   union REGS regs;

    if (No > 128)
       {
        cmf_err_stat = 400;
        return(FALSE);
        };
    regs.x.bx = 2;
    regs.x.cx = No;
    regs.x.dx = FP_SEG(cmf_instrument);
    regs.x.ax = FP_OFF(cmf_instrument);
    int86(cmf_driverirq, &regs, &regs);
    return(TRUE);
   };

void _cmf_set_sysclock(WORD frequency)
/*
 * INPUT   : System timer clock rate as WORD
 * OUTPUT  : none
 * PURPOSE : Sets the timer default value of 0 to a new value.
 */

{
   union REGS regs;
   ldiv_t dummy;

   dummy = ldiv(1193180L, (long)frequency);
   regs.x.bx = 3;
   regs.x.ax = dummy.quot;
   int86(cmf_driverirq, &regs, &regs);
   };

void _cmf_set_driverclock(WORD frequency)
/*
 * INPUT   : Timer clock rate as WORD
 * OUTPUT  : none
 * PURPOSE : Sets the timer frequency for driver to a new value.
 */

{
   union REGS regs;
   ldiv_t dummy;

   dummy = ldiv(1193180L, (long)frequency);
   regs.x.bx = 4;
```

```
   regs.x.ax = dummy.quot;
   int86(cmf_driverirq, &regs, &regs);
   };

void _cmf_set_transposeofs (int offset)
/*
 * INPUT   : Offset as WORD. The value specifies the number of half
             steps by which the notes should be transposed
 * OUTPUT  : none
 * PURPOSE : Transposes all the notes that are played by "Offset".
 */

{
   union REGS regs;

   regs.x.bx = 5;
   regs.x.ax = offset;
   int86(cmf_driverirq, &regs, &regs);
   };

BYTE _cmf_play_song(CMFPTR cmf_buffer)
/*
 * INPUT   : Pointer to the song data
 * OUTPUT  : TRUE, if start is successful, otherwise FALSE
 * PURPOSE : Initializes all important parameters and starts song output
 */

{
   BYTE check;
   union REGS regs;

/* Sets the driver clock frequency */
   _cmf_set_driverclock((WORD)cmf_buffer[CTPS]);
/* Sets the instruments */
   check = _cmf_set_instruments(cmf_buffer);
   if (check == FALSE) return(FALSE);

   regs.x.bx = 6;
   regs.x.dx = FP_SEG(cmf_buffer);
/* This strange typecasting was necessary because of a compiler problem */
/* Ordinarily WORD typecasting is enough                                */
   regs.x.ax = (BYTE)cmf_buffer[MUSOFF]+256*(BYTE)cmf_buffer[MUSOFF+1];
   int86(cmf_driverirq, &regs, &regs);

   if (regs.x.ax == 0)
     {
      cmf_songpaused = FALSE;
      return(TRUE);
      }
   else
     {
      cmf_err_stat = 500;
      return(FALSE);
      };
```

```
   };

BYTE _cmf_stop_song(void)
/*
 * INPUT   : None
 * OUTPUT  : TRUE/FALSE, depending on whether or not song stops
 * PURPOSE : Tries to stop a song.
 */

{
   union REGS regs;

   regs.x.bx = 7;
   int86(cmf_driverirq, &regs, &regs);
   if (regs.x.ax == 0) return(TRUE);
   else
     {
      cmf_err_stat =  510;
      return(FALSE);
      };
   };

BYTE _cmf_reset_driver(void)
/*
 * INPUT   : none
 * OUTPUT  : none
 * PURPOSE : Resets the driver to start status.
 */

{
   union REGS regs;

   regs.x.bx = 8;
   int86(cmf_driverirq, &regs, &regs);
   if (regs.x.ax == 0) return(TRUE);
   else
     {
      cmf_err_stat = 110;
      return(FALSE);
      };
   };

BYTE _cmf_pause_song(void)
/*
 * INPUT   : None
 * OUTPUT  : TRUE/FALSE, depending on whether or not song pauses
 * PURPOSE : Tries to pause a song. If this is possible, then the global
             variable cmf_songpaused is set to TRUE.
*/

{
   union REGS regs;

   regs.x.bx = 9;
```

```
   int86(cmf_driverirq, &regs, &regs);
   if (regs.x.ax == 0)
     {
      cmf_songpaused = TRUE;
      return(TRUE);
      }
   else
     {
      cmf_err_stat = 520;
      return(FALSE);
      };
   };

BYTE _cmf_continue_song(void)
/*
 * INPUT   : None
 * OUTPUT  : TRUE/FALSE, depending on whether or not song continues
 * PURPOSE : Tries to continue a song. If this is possible, the global
             variable cmf_songpaused is set to FALSE.
 */

{
   union REGS regs;

   regs.x.bx = 10;
   int86(cmf_driverirq, &regs, &regs);
   if (regs.x.ax == 0)
     {
      cmf_songpaused = FALSE;
      return(TRUE);
      }
   else
     {
      cmf_err_stat = 530;
      return(FALSE);
      };
   };

BYTE _cmf_prepare_use(void)
/*
 * INPUT   : None
 * OUTPUT  : TRUE or FALSE, depending on success of initialization
 * PURPOSE : Initializes the driver and sets the _cmf_status_byte
             address for the driver.
 */
{
  if (_cmf_init_driver() == TRUE)
    {
     _cmf_set_status_byte();
     return(TRUE);
    }
  return(FALSE);
}
```

```
void _cmf_terminate_use(void)
/*
 * INPUT   : None
 * OUTPUT  : None
 * PURPOSE : Resets the StatusByte address, so that the driver doesn't
             write to any memory areas after the end of the program.
 */
{
  union REGS regs;

  regs.x.bx = 1;
  regs.x.dx = 0;
  regs.x.ax = 0;
  int86(cmf_driverirq, &regs, &regs);
}
```

CMFDEMO

We'll use a sample program to demonstrate how the functions in CMFTOOL.H can be used. This example is located in the CMFDEMO.C file. The companion diskette includes both source codes and an executable version of CMFDEMO.EXE.

Running CMFDEMO

Make sure that the STARFM.CMF file is in the same directory as CMFDEMO.EXE. Run SBFMDRV.COM, then run CMFDEMO.EXE.

The textnumerror() function

As in the previous examples, the textnumerror() function displays an appropriate error number and text on the screen and then ends the program with the corresponding error level.

Main program

The main program prepares the songbuffer variable for loading the CMF data. The character string "STARFM.CMF" is assigned to the songname variable.

STARFM.CMF is one of the demo songs included on the Sound Blaster card diskettes. You can also use the name of any other CMF file instead of STARFM.CMF, either added to the program code or entered as a parameter when you call CMFDEMO.EXE.

After the screen is cleared, the program checks whether the driver is actually resident in memory. If it isn't, the program is immediately terminated with the textnumerror() function.

CMF version numbers

If the initialization has been performed successfully, the driver's version number is displayed.

Don't be surprised if the number that's displayed differs from the version number that was displayed when you ran SBFMDRV.COM.

Different numbers are used for the internal version number and the number that's displayed at startup. However, this doesn't mean that function 0 has been used incorrectly.

Next the number of the interrupt that's being used and other information is displayed in hexadecimal format.

Then the specified file is loaded into memory, and the resulting function value, a pointer identifying the data that have just been loaded, is assigned to songbuffer. If the value of this pointer is ZERO, the program is terminated with the textnumerror() function.

The next program line allows you to shift the pitch of the song up or down. The value "0" used in the example indicates that the data will be played back without being transposed. Try experimenting with different values.

Then the playback is started with _cmf_play_buffer(songbuffer). If the start is successful, the program execution continues. Otherwise it's terminated with the textnumerror() function.

During the playback, the value of cmf_status_byte is continually displayed. The program remains in this loop until the content of this variable is "0", which indicates that the playback is complete.

Another way of ending the playback is to press a key. Then the loop is interrupted and _cmf_stop_song() is called.

After the playback has been completed, some additional tasks still must be performed. First the FM registers are reset and then the memory occupied by the CMF data is freed.

Finally, _cmf_terminate_use() is called, after which the program can be ended.

This sample program is very simple. In your own programs you'll be able to add your own functions and have wider control of the FM synthesis hardware.

Although there are many CMF files, you can also use the ROL2CMF program, available as shareware, to make AdLib ROL files accessible to your programs.

```
/*
 **************************************************************************
 *   Demonstration program  for CMFTOOL module, (W) in Borland C++ 3.1   *
 **************************************************************************
 *                           (C) 1992 Abacus                              *
 *                          Author : Axel Stolz                           *
 **************************************************************************
 */

#include <conio.h>
#include "cmftool.h"

void textnumerror()
/*
 * INPUT   : None; data comes from the cmf_err_stat global variable
 * OUTPUT  : None
 * PURPOSE : Displays SB errors as text on the screen. Includes
             error number and ends program with the error level that
             corresponds to the error number.
 */
{
   printf(" Error # %d, ",cmf_err_stat);
   _print_cmf_errmessage();
   exit(cmf_err_stat);
}


void main(void)
{
 CMFPTR songbuffer;
 char   songname[] = "starfm.cmf\0";

 clrscr();

/* If the SBFMDRV driver is not installed, display error,  */
/* otherwise the driver will be initialized                */
 if (_cmf_prepare_use() == FALSE) textnumerror();

/* Display the main version and subversion number of the driver */
 gotoxy(28,5);
 printf("SBFMDRV Version ");
 printf("%d.%02d ",_cmf_getversion() >> 8, _cmf_getversion() % 256);
 printf("loaded");

/* Display interrupt number currently in use */
 gotoxy(24,10);
 printf("System interrupt (IRQ) 0x%x in use\n",cmf_driverirq);
 gotoxy(35,15);
 printf("Song Status");
 gotoxy(31,23);
 printf("Song name: %s\n",songname);

/* Load the desired song file */
 songbuffer = _cmf_get_songbuffer(songname);
```

```
 if (songbuffer == NULL) textnumerror();

/*
 You can transpose the loaded piece of music either up or down,
 depending on whether you pass a positive or a negative value.
 0 plays the piece of music in the original key.
*/
 _cmf_set_transposeofs(0);

/* Keep displaying status byte during playback */
 if (_cmf_play_song(songbuffer) != TRUE) textnumerror();
 do
   {
    gotoxy(39,17);
    printf("%3d",cmf_status_byte);
   } while ((cmf_status_byte != 0) && (kbhit() == 0));

/* If key is pressed, then interrupt the song */
 if (kbhit() != 0) if (_cmf_stop_song() != TRUE) textnumerror();

/* Transfer driver to enable status */
 if (_cmf_reset_driver() != TRUE) textnumerror();

/* Release memory of loaded song file */
 _dos_freemem(FP_SEG(songbuffer));

/* Switch off CMFTool functions */
 _cmf_terminate_use();
}
```

5.4 Soundtracker Format

Amiga MOD format

The Soundtracker format is probably the most popular music file format currently used on Amiga systems. Soundtracker modules are found in games, demos, and many other types of programs that rely on some kind of sound.

Because of its popularity, this format has been constantly modified to try to make it more efficient and powerful. However, since these changes occurred gradually, there are several versions of the Soundtracker format. So, different names were given to each of these formats, such as Noisetracker, Protracker, and Startracker.

Short but impressive

The basic concept of the Soundtracker format is to use digitized instrument samples and to bring these samples to the desired pitch by modifying their playback frequency.

This makes it possible to create very realistic musical pieces because sampled instruments sound much more realistic than

synthesized ones. Also, since each instrument must be loaded into memory only once, this results in much shorter files than fully digitized pieces of music. So you can create long songs without a lot of information.

In addition to modifying an instrument's pitch, it's also possible to change its sound in other ways, for example by adding vibrato.

The original Soundtracker format could handle 15 different digital instruments. However, this was soon expanded to 31. So, in this section we'll discuss the 31 instrument format.

Normal Soundtracker music is designed for four channels because the Amiga has four sound channels. However, newer versions easily produce eight independently programmable channels on the Amiga. This means that two programmable channels are played on each actual sound channel.

Four into one

This becomes even more complicated when the Soundtracker module is transferred to the Sound Blaster card because this card has only one digital channel, except for SB Pro. This means that four Amiga channels must be combined into one Sound Blaster channel.

On the Amiga, two channels are fed into the left stereo channel and two are fed into the right. However, this effect is also lost when the Sound Blaster card is used.

Special MOD players are available for only the SB Pro. These players feed the four Amiga channels into the two SB Pro channels. (For more information about MOD file players refer to Chapter 2.)

Now let's return to the Amiga. The notes for Soundtracker songs are organized in up to 64 patterns. Each pattern contains 4 tracks for the four channels, which each have 64 notes.

This can be confusing because the names for tracks and patterns are switched in many PC MOD players. So a unit of 64 notes for the four channels is often referred to as a track instead of a pattern.

However, we'll use the first definition. A pattern might look as follows:

Note	Track1	Track2	Track3	Track4
0	C-1(1)	C-1(1)	C-2(2)	C-2(2)
1	D-1(1)	D-1(2)	D-2(4)	D-2(4)
2	E-1(1)	E-1(1)	C-2(2)	C-2(2)
...	...	...	...	...
63	F#1(5)	F#3(7)	A-1(1)	D-1(1)

The number for each note within the pattern is listed under "Note." Under "Tracks" you'll see the corresponding value for each note and each channel, along with the instrument number in parentheses.

The patterns defined in this way can then be added in any sequence to create a piece of music. For example, if the refrain of your song consists of the four patterns "14, 15, 16, 17", you can insert these four patterns in the places in the song where the refrain should appear.

This saves a considerable amount of memory because the refrain doesn't need to be reprogrammed for each time it's played. The patterns are arranged similar to the method used on VOC files in the sample program ARRANGER, which appeared in the section on the digital sound channel.

This means that a pattern doesn't have to be loaded into memory more than once.

The MODEDIT program, which is described in the section on shareware and public domain programs, is ideal for experimenting with these patterns.

In the following section we'll discuss the structure of a MOD file in detail.

5.4.1 Structure of a MOD file

The MOD file structure described below corresponds to a recent version of the Soundtracker format. So it's possible that your older modules may not match this format byte for byte.

However, the largest portion of the module, with 31 instruments and the ones with 15 instruments, can be explored as follows:

Bytes $00 - $13 (0 - 19)

Song name

The first 20 bytes contain the song's name as ASCII text. The last byte must contain a 0 to indicate the end of the text.

Bytes $14 - $31 (20 - 49)

Instrument 1

These bytes contain the information for the first digitized instrument. Each instrument requires 30 bytes of information, which are described below.

Bytes $14 - $29 (20 - 41)

Instrument name

These 22 bytes contain the name for instrument 1 as ASCII text. Again, the name must be terminated with a 0 byte.

Bytes $2A - $2B (41 - 43)

Instrument length

These bytes contain the length of the first instrument as a number of words (16-bit).

If you want to read these values, remember that you're working with an Amiga file. The 68000 processor manages its word values (two bytes) in a different sequence than the Intel processors. The 68000 switches the sequence of high-byte/low-byte. So you must switch this sequence back to receive the correct values. The same applies to all other word values in this section.

The value that you'll receive after switching the high and low bytes must then be multiplied by 2 to obtain the actual length of the instrument in bytes.

Byte $2C (44)

Fine tune

Only the lower four bits of this byte are used. The upper four bits should always be 0. The lower four bits specify a value between -8 and 8, in the form of a 2's complement, which is used to fine tune the pitch of the instrument within certain limits.

Byte $2D (45)

Loudness

This byte specifies the instrument's loudness. Values between 0 and 64 are permitted.

Bytes $2E - $2F (46 - 47)

Repeat loop

Soundtracker allows you to repeat an instrument an unlimited number of times, until another instrument is started on that channel. This effect can be used, for example, to create an echo or

reverberation. These two bytes specify at which point the repetition should begin.

The value of this byte specifies the number of words (16-bit) measured from the start of the instrument data.

Bytes $30 - $31 (48 - 49)

Repeat length

These two bytes contain the length of the repetition as a 16-bit word. So, with the bytes $2E and $2F, you now know where the repetition of an instrument sound should begin and how long the repetition should be.

This concludes the information for an instrument. With a total of 31 instruments, 30 more identical blocks will follow, one for each of the 30 remaining instruments. With a total of 15 instruments, there will be 14 remaining blocks.

The bytes listed in the following section are numbered according to a 31 instrument file. So we'll continue with byte number $3B6 (950).

Byte $3B6 (950)

Song length

This byte contains the number of patterns that must be played within the entire song. However, this isn't necessarily the actual number of patterns in the song.

For example, if you've defined 42 different patterns for your piece of music, but have chosen an arrangement consisting of "07 12 05 05 07 23", then this byte will contain the value 6.

Byte $3B7 (951)

CIAA speed

Since this byte determines CIAA speed, it's used mainly for the Amiga. So it isn't very important on PCs.

This value is usually 127 ($7F). However, occasionally this byte is used for other purposes. So it's difficult to determine what each value will indicate.

Bytes $3B8 - $437 (952 - 1079)

Song arrangement

These 128 bytes contain the order in which the song's patterns are played back. Each byte can receive a value between 0 and 63. This number corresponds to the desired pattern number. So these 128 bytes determine the song's arrangement.

Bytes $438 - $43B (1080 - 1083)

Marking files

In MOD files with 31 instruments, these four bytes contain the text "M.K." or "FLT4". So you can identify the file as a 31-instrument file.

If this text isn't found at this location, the file is either a module, from which the text has been purposely removed (e.g., to prevent the song from being "stolen" by other programs) or it's a file with only 15 instruments. MOD files with 15 instruments don't contain these four identification bytes.

Bytes $43C - $83B (1084 - 2107)

Patterns

These are 1024 bytes that contain the 64 notes of a pattern. Each note consists of 4 bytes. The four notes for the four individual channels are always stored consecutively. So each line of the pattern is 16 (4*4) bytes long. Since a pattern contains 64 lines, it contains a total of 1024 (64*16) bytes.

This structure of 4 bytes per note is the biggest difference between 31-instrument files and 15-instrument files.

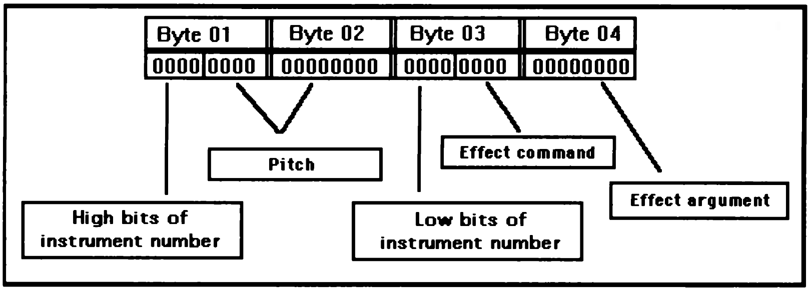
With the Amiga, these 4 bytes are stored in the form of a longword (i.e., a positive 32-bit number). This is a different sequence than the Intel convention. Remember this difference if you intend to access these values with your own programs.

Structure of MOD notes

In 15-instrument files, bits 0-3 of the LONG WORD still belonged to the song pitch, and bits 16-19 contained the instrument number. This is the reason for the 15 instrument limit. So bits 0-3 are used as the upper 4 bits of the instrument number and bits 16-19 still act as the lower 4 bits of the instrument number. Theoretically this allows for more than 31 instruments, but few MOD files take advantage of more than 31 instruments.

Bits 4 through 15 specify the song pitch. This value indicates the number of samples per minute. Bits 20-23 contain an effect number between 0 and 15 and bits 24-31 contain the operands for this effect.

The following figure shows how the four bytes of a note are combined:



The structure of a MOD note

MOD effects

The effect commands can be used to assign specific qualities to the notes that are played back. An effect number can have either one or two arguments. In the first case, bits 24-31 are interpreted as one value (xxx). In the latter case, bits 24-27 are used as the first operand (x) and bits 28-31 as the second (y).

Effect 00 - Arpeggiation:xy

Effect command 00 produces an arpeggio for this note. This means that the tone is played at three different pitches in rapid succession; x specifies the first interval and y specifies the second. Both arguments specify the number of half-steps that the interval must span.

Effect 01- Slide Up:xx

This effect increases a tone's pitch during playback. The argument "xx" specifies the rate at which the tone's pitch must be increased.

Effect 02 - Slide Down:xx

This is the same effect as number 01, except that this effect decreases a note's pitch during playback.

Effect 03 - Slide To Note:xx

This effect allows a tone to be raised or lowered to a specified pitch. Unlike effect numbers 1 and 2, this effect allows you to specify the final pitch at which the tone will come to rest. The argument "xx" again specifies the rate at which the tone's pitch is changed.

Effect 04 - Vibrato:xy

Effect 04 produces a vibrato for the corresponding note. Argument "x" specifies the vibrato's rate, while "y" indicates its depth.

Effect 10 - Volume Slide:x0 or 0y

This effect allows you to increase or decrease a note's loudness. An argument of type "x0" increases the loudness at rate "x", while an argument of type "0y" decreases the loudness at rate "y".

Effect 11 - Position Jump:xx

This effect stops the playback of the current pattern and causes the playback procedure to continue at location "xx" of the pattern arrangement. Therefore "xx" may only range from 0 to 127.

Effect 12- Set Volume:xx

This effect sets a note's loudness to the value of "xx". Although this effect isn't actually intended for musical passages of a song, it's useful for combining various sound effects with a song. For example, you may want to add the sound of a starting car to a song at a higher or lower loudness level than the other channels.

However, if you want to use this effect on each note of a passage, you'll be able to create characteristic pieces of music through the loudness variation created in this way. The argument value "xx" may range from 0 to 64.

Effect 13 - Pattern Break:00

This effect command stops the playback of the current pattern and continues with the playback of the next pattern. The value of the argument byte for this effect is always 0.

Effect 15 - Set Speed:xx

This effect sets the playback speed for the specified song. Values between 0 and 31 are permitted for the argument "xx".

All effect numbers that haven't been listed here aren't used. However, some playback procedures may be equipped with several additional effects.

So if you want to create your own playback procedure, simply ignore all unrecognized effect commands. Now let's return to the structure of the MOD format.

Bytes $83C - ... (2108 - ...)

The data for the other defined patterns begins here. The number of patterns a MOD file contains can be determined only through a complex calculation.

First you must calculate the total length of all instrument data (i.e., the sample information). The necessary value can be found through the corresponding instrument information contained in the MOD header.

Don't forget that the length is specified in WORDs. So you must double this value to receive the length in bytes.

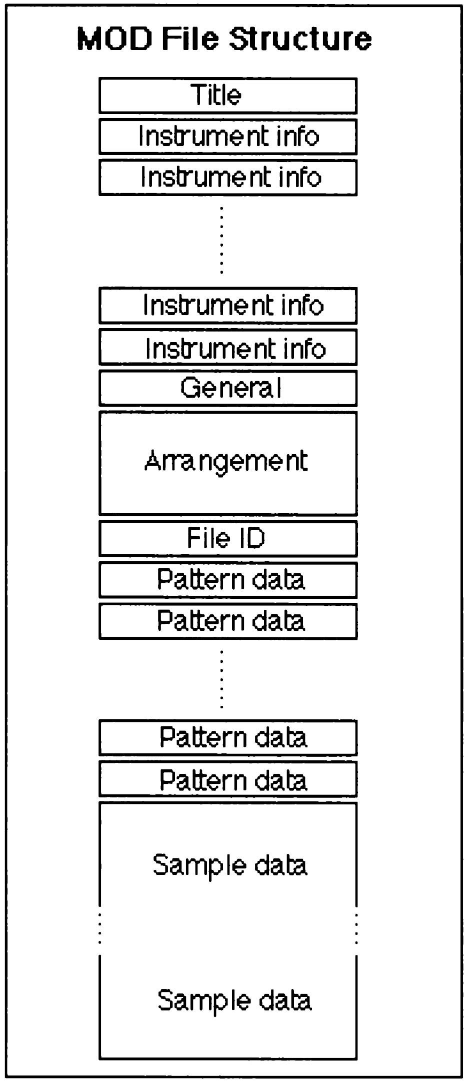
Then the number of bytes in the header is added. This is the total number of bytes before the start of the pattern data.

Then this sum must be subtracted from the total length of the file, leaving you with the length of the pattern information.

Since each pattern consists of 1024 bytes, simply divide this number by 1024 to obtain the number of actual patterns contained by the module file.

The pattern information is followed by the instrument data in the form of raw 8-bit data. Where each individual instrument begins and ends is indicated by the information in the MOD header.

The structure of a MOD file looks as follows:



Overview of a MOD file's structure

5.4.2 The MODSCRIPT program

To help you understand the structure of MOD files, we've included a program called MODSCRIP.PAS on the companion diskette. This program reads all pertinent information from a 31-instrument file and displays the file's patterns and note values.

MODTOOL unit

The program was written in Turbo Pascal 6.0 using the MODTOOL.PAS unit. The structure of a MOD file is implemented within this unit.

The MOD data types

The InstrumentType record contains the data structure needed for the instrument information found in the MOD header. The data type MODHeaderType contains the entire file header for a 31-instrument file.

The data type NoteType is defined as a four-element ARRAY...OF BYTE because this allows easy access to each individual bit of the note.

The type PatternLine contains four notes (one for each channel) and the type PatternType contains an entire pattern (i.e., 64 pattern lines).

Function NoteName

The Octaves constant contains a tuning table for 36 notes (i.e., three octaves from C1 to C4). Each table value corresponds to a half-step ranging from the lowest tone (C1) to the highest (B3).

You can use this table to convert the pitch values stored in each note to the corresponding note name.

This task is performed by the NoteName function. The number that's contained in bits 4-15 of the note is passed to the function as its parameter and the function returns a string value with the name of the specified note.

Function Words2Bytes

As we mentioned, the 68000 processor stores its word values in the opposite sequence as Intel processors. So you must always switch the bytes of word values read from any MOD header before processing this data (e.g., when you're trying to determine the length of an instrument).

For this purpose, MODTOOL provides the Words2Bytes function. Simply pass the "backward" word to the function. You'll receive the value that was contained in this word multiplied by 2, since this function is used to determine byte-lengths.

Otherwise, simply use the Swap command to switch the WORD's high and low bytes.

Procedure BuildModScript

BuildModScript performs the primary tasks of MODTOOL.TPU. It produces a text file that contains the most important information on the MOD file.

This procedure is an example of how you can access the information contained in a MOD file by using the data structures previously described.

You must specify two parameters: the name of the MOD file and the name of the text file that will be created.

First the MOD file is opened, if it actually exists, and then the entire header is loaded to a variable of type MODHeaderType. Then the specified text file is created and a heading is written to the file, followed by the song name and the number of patterns contained in the arrangement.

Next the highest pattern number appearing in this arrangement is determined.

However, the procedure doesn't determine the actual number of patterns contained in the file. The number of patterns that are considered is limited by the highest pattern number in the file. So if you've defined 47 patterns within a MOD file, but the highest pattern number is 42, then patterns 43 through 47 are ignored by BuildModScript.

Instrument data

Next the instrument data for all 31 instruments are read. For each instrument, this includes the instrument name, the length in bytes, the fine tune factor, the loudness, the repeat start in bytes, as well as the repeat length in bytes.

Pattern data

After the instrument information, all patterns from number 0 to the highest pattern number are displayed in text format. This means that, for each pattern line, the corresponding note is listed for each of the four channels. Also, the number of the instrument used for that note is listed within parentheses.

This portion of the procedure shows how to obtain the information, indicating a note's pitch and instrument number, through its four note bytes.

The individual effects are ignored. However, it's easy to determine the effect number and argument for each note.

Once all patterns have been displayed, the MOD file and the text file are closed.

The capabilities of this unit are used in the MODSCRIP.PAS program.

```
Unit MODTool;
{
 ********************************************************************
 *    Unit for reading information from the Soundtracker module.    *
 ********************************************************************
 *                        (C) 1992 Abacus                           *
 *                      Author : Axel Stolz                         *
 ********************************************************************
 * This routine expects a module containing 31 instruments. If this *
 * is not the case, the result may be somewhat different.           *
 ********************************************************************
}

INTERFACE

CONST
   MODToolVersion = 'v1.0';

   MaxIns = 31;  { Maximum number of instruments in module }

   Octaves : ARRAY[1..36] OF WORD =      { Tuning table for 3 octaves }
      (856,808,762,720,678,640,604,570,538,508,480,453,
       428,404,381,360,339,320,302,285,269,254,240,226,
       214,202,190,180,170,160,151,143,135,127,120,113);

TYPE
   InstrumentType = RECORD
      SampName : ARRAY[0..21] OF CHAR;  { Instrument name             }
      SampLen  : WORD;                  { Instrument length in words }
      SampTune : BYTE;                  { Finetune options            }
      SampAmp  : BYTE;                  { Instrument amplitude (vol) }
      SampRepS : WORD;                  { Repeat start in words      }
      SampRepL : WORD;                  { Repeat length in words     }
      END;

   MODHeaderType = RECORD
      MODName  : ARRAY[0..19] OF CHAR;                 { MOD filename }
      MODInstr : ARRAY[1..MaxIns] OF InstrumentType;  { Instruments  }
      MODLen   : BYTE;                                 { Song length  }
      MODMisc  : BYTE;                                 { CIAA speed   }
      MODPattr : ARRAY[1..128] OF BYTE;                { Pattern list }
      MODSign  : ARRAY[1..4] OF CHAR;                  { Sign string  }
      END;

   NoteType    = ARRAY[1..4] OF BYTE;         { One note  = 4 bytes    }

   PatternLine = RECORD                       { 1 pattern line set }
```

```
      Channel1, Channel2,                    { for 4 channels     }
      Channel3, Channel4   : NoteType;
      END;

   PatternType = ARRAY[1..64] OF PatternLine; { One pattern = 64 notes }


PROCEDURE BuildModScript(ModFilename, ScriptFilename : STRING);

IMPLEMENTATION

FUNCTION ConvertString(Source : Pointer; Size : BYTE):String;
{
 * INPUT   : Pointer to an ARRAY OF CHAR, length in BYTES
 * OUTPUT  : Pascal string in Size bytes length
 * PURPOSE : Convertor, e.g., converts string ending in NULL to a Pascal
              string. This routine can convert any other memory range
              into a string.
}
VAR
   WorkStr : String;
BEGIN
   Move(Source^,WorkStr[1],Size);
   WorkStr[0] := CHR(Size);
   ConvertString := WorkStr;
   END;

FUNCTION Words(FalseWord : WORD):WORD;
{
 * INPUT   : Word variable with rotated high-byte/low-byte
 * OUTPUT  : Restored Word, multiplied by 2
 * PURPOSE : Gets a Word value from the MOD file and restores the proper
             high-byte/low-byte sequence; also multiplies the result by
             2, as this function must store the information as Word units
             to coincide with sample length.
}
BEGIN
   Words := (Hi(FalseWord)+Lo(FalseWord)*256)*2;
   END;

FUNCTION NoteName(Period : WORD):String;
{
 * INPUT   : Note length value as WORD
 * OUTPUT  : Note in text as string
 * PURPOSE : Using tuning table, converts pitch
             values into the proper note names.
}

CONST
   NNames : ARRAY[0..11] OF String[2] =
      ('C-','C#','D-','D#','E-','F-','F#','G-','G#','A-','A#','B-');
VAR
   WorkStr : String;
   NCount  : BYTE;
```

```
BEGIN
   NCount := 1;
   IF (Period = 0) THEN BEGIN
      NCount := 37;
      NoteName := '----';
      END;
   WHILE (NCount <= 36) DO BEGIN
      IF (Period = Octaves[NCount]) THEN BEGIN
         Dec(Ncount);
         Str((NCount DIV 12)+1:2,WorkStr);
         NoteName := NNames[NCount-(NCount DIV 12)*12]+WorkStr;
         NCount := 37;
         END;
      Inc(NCount);
      END;
   END;

PROCEDURE BuildModScript(ModFilename, ScriptFilename : STRING);
{
 * INPUT   : Module name and name of desired script file
 * OUTPUT  : None
 * PURPOSE : Reads a Soundtracker module and writes the most important
             information to a text file.  The module must contain 31
             instruments, or incorrect results will occur.
             Patterns are stored in sequence.
}

VAR
   ModFile  : File;          { File variable for ST module            }
   ScrFile  : TEXT;          { Text file variable for script file     }
   Result   : WORD;          { Dummy variable for Blockread           }
   Header   : ModHeaderType; { Variable for MOD header info           }
   DummyStr : String;        { String variable for data conversion    }
   InsCount : BYTE;          { Counter variable for instrument number }
   PatCount : BYTE;          { Counter variable for pattern number    }
   Pattern  : PatternType;   { Buffer variable for a pattern          }
   HiPatt   : BYTE;          { Highest pattern number in module       }
   Counter  : WORD;          { General counter variable               }

BEGIN
{ Make sure that desired MOD file is available }
{$I-}
   Assign(ModFile,ModFilename);
   Reset(ModFile);
   Close(ModFile);
{$I+}
   IF (IOResult <> 0) THEN BEGIN
      Writeln('Cannot find MOD file: ',ModFilename:12,'. Sorry.');
      HALT(100);
      END;

{ Read MOD header data into Header variable }
   Reset(ModFile,1);
   BlockRead(ModFile,Header,SizeOf(Header),Result);
```

```
{ Analyze data in Header }
   WITH Header DO BEGIN
      Assign(ScrFile,ScriptFilename);
      ReWrite(ScrFile);
      WriteLn(ScrFile,'Soundtracker module script file ',ModFilename);
      WriteLn(ScrFile);

{ Write module name }
      DummyStr := ConvertString(Addr(MODName),SizeOf(MODName));
      WriteLn(ScrFile,'Module name  : ',DummyStr);

{ Write module length (valid numbers = 1 - 128) }
      Str(MODLen,DummyStr);
      Write  (ScrFile,'Module length  : ',DummyStr, ' Pattern(s),');

{ Search for highest pattern number }
      HiPatt := 0;
      FOR PatCount := 1 TO MODLen DO
         IF ModPattr[PatCount] >= HiPatt THEN
            HiPatt := ModPattr[PatCount];
      Str(HiPatt,DummyStr);
      WriteLn(ScrFile,' - highest pattern number is ',DummyStr);
      WriteLn(ScrFile);

{ Write instrument information }
      FOR InsCount := 1 TO MaxIns DO BEGIN
         WITH MODInstr[InsCount] DO BEGIN
            DummyStr := ConvertString(Addr(SampName),SizeOf(SampName));
            WriteLn(ScrFile,'Instrument # ',InsCount:2,' = ',DummyStr);
            Str(Words(SampLen):6,DummyStr);
            WriteLn(ScrFile,'Length in bytes = ',DummyStr);
            Str(SampTune:6,DummyStr);
            WriteLn(ScrFile,'Fine tune       = ',DummyStr);
            Str(SampAmp:6,DummyStr);
            WriteLn(ScrFile,'Volume          = ',DummyStr);
            Str(Words(SampRepS):6,DummyStr);
            WriteLn(ScrFile,'Repeat start    = ',DummyStr);
            Str(Words(SampRepL):6,DummyStr);
            WriteLn(ScrFile,'Repeat length   = ',DummyStr);
            WriteLn(ScrFile,'---------------------------------------');
            END;
      END;

{ Read patterns and write note values into script file }
{ (parentheses following pitch name contain sample number of note) }
      FOR PatCount := 1 TO HiPatt+1 DO BEGIN
         IF NOT(EOF(ModFile)) THEN
            Blockread(ModFile,Pattern,SizeOf(Pattern),Result);

         WriteLn('Read pattern ',PatCount-1:3);

         WriteLn(ScrFile,'Pattern number : ',PatCount-1:3);
```

```
         WriteLn(ScrFile,'Lines #   Chan.1     Chan.2     Chan.3
Chan.4');
         FOR Counter := 1 TO 64 DO BEGIN
            Write(ScrFile,'  ',Counter:2,'    ');
            WITH Pattern[Counter] DO BEGIN

{ Display note and sample number for channel 1 }
               DummyStr := NoteName((Channel1[1] AND $0F)*256+(Channel1[2]));
               Write(ScrFile,' ',DummyStr);
               Write(ScrFile,'(',((Channel1[1] AND $F0)+
                                   (Channel1[3] SHR 4)):2,')  ');

{ Display note and sample number for channel 2 }
               DummyStr := NoteName((Channel2[1] AND $0F)*256+(Channel2[2]));
               Write(ScrFile,' ',DummyStr);
               Write(ScrFile,'(',((Channel2[1] AND $F0)+
                                   (Channel2[3] SHR 4)):2,')  ');

{ Display note and sample number for channel 3 }
               DummyStr := NoteName((Channel3[1] AND $0F)*256+(Channel3[2]));
               Write(ScrFile,' ',DummyStr);
               Write(ScrFile,'(',((Channel3[1] AND $F0)+
                                   (Channel3[3] SHR 4)):2,')  ');

{ Display note and sample number for channel 4 }
               DummyStr := NoteName((Channel4[1] AND $0F)*256+(Channel4[2]));
               Write(ScrFile,' ',DummyStr);
               Write(ScrFile,'(',((Channel4[1] AND $F0)+
                                   (Channel4[3] SHR 4)):2,')  ');
               WriteLn(ScrFile);
               END;
            END;
         WriteLn(ScrFile,'-----------------------------------------------');
         WriteLn(ScrFile);
         END;

{ Close script file and MOD file }
      Close(ScrFile);
      CLose(ModFile);
      END;
   END;
END
```

MODSCRIP

When you call the MODSCRIP.EXE file, you must include the name of the MOD file, with or without .MOD extension. The following will work for the TESTSONG.MOD file included on the companion diskette:

```
modscrip testsong.mod [Enter]
modscrip testsong [Enter]
```

If you do not include a MOD filename as a parameter, MODSCRIP displays a help text.

Then the BuildModScript procedure, which creates the text file for the specified MOD file, is called. The name of the MOD file is also used as the name for the text file, except that the filename extension .TXT is used.

This program enables you to create a script file, from most Soundtracker modules with 31 instruments, through a simple program call at the DOS command line.

However, the program's main purpose is to illustrate how the data in a MOD file are accessed.

```
Program MODScript;
{
 *********************************************************************
 * Demonstration program for MODTOOL unit: Creating a script file   *
 *********************************************************************
 *                        (C) 1992 Abacus                           *
 *                       Author: Axel Stolz                         *
 *********************************************************************
 * ModScript creates a FILE.TXT file from the Soundtracker module   *
 * 'FILE.MOD,' which describes all instruments, instrument          *
 * parameters, and patterns with their respective note contents.    *
 *********************************************************************
}

Uses MODTool; { Use MODTool unit }

VAR
   WorkStr : String; { String for processing provided filename }

BEGIN
   WriteLn('MODScript v1.0 (C) 1992 Abacus -  Author: Axel Stolz');

{ No command line parameter given? display syntax }
   IF (ParamCount = 0) THEN BEGIN
      WriteLn('Syntax : MODSCRIP module[.MOD]');
      HALT(0);
      END;

{ Pass command line parameters }
   WorkStr := ParamStr(1);

{
  If user enters a filename and extension, replace extension
  with ".MOD" extension
}
   IF Pos('.',WorkStr) > 0 THEN
      WorkStr := Copy(WorkStr,1,Pos('.',WorkStr)-1);
```

```
  WriteLn('Create a script file from a sound module ',WorkStr,'.MOD !');

{ Create script file }
  BuildModScript(WorkStr+'.MOD',WorkStr+'.TXT');
  WriteLn('Ready. Result stored in ',WorkStr,'.TXT.');
  END.
```

You can write your own MOD editor based on this program code, or develop a program that converts MOD files to other file formats. You could also transform MOD files into standard MIDI files.

5.5 MIDI File Format

While working with Sound Blaster, you'll encounter files with the .MID extension.

These files are used under Windows and can be processed by the Voyetra Sequencer program series. Newer software packages, that are shipped with the Sound Blaster card, include the program PLAYMIDI.EXE (in addition to PLAYCMF.EXE), which can play back these files.

All these programs are capable of reading standard MIDI files (SMF).

5.5.1 Structure of MIDI files

As presented in Chapter 4, the MIDI language follows a specific order so various devices can access these files. Because of this, a file standard, on which several programs could be based, was eventually developed.

The SMF format was developed so various programs could exchange MIDI data on different types of computers. However, this format was developed much later than the actual MIDI language. So the first versions were written without the approval of the International MIDI Association (IMA). However, the format was soon corrected and standardized.

IFF standard influence

The structure of SMF files is mainly based on the idea that Electronic Arts introduced with their IFF standard.

The data are divided into chunks, which consist of a 32-bit name and a 32-bit length value. This is followed by the actual chunk data. You may remember this procedure from the discussion of RIFF files.

So SMF files are as flexible. Also, you can easily expand this format without encountering compatibility problems with older or newer versions.

Unrecognized chunk types are simply skipped and recognized ones are processed.

Header + Track = SMF

Until now only two different chunk types have been created: Header chunks and track chunks. The header chunk is identified by "MThd" and the track chunk is identified by "MTrk".

A header must appear at the beginning of each file, followed by the tracks.

The SMF also doesn't have chunks that can contain another chunk. So constructions such as "track in track" aren't possible. Instead there are three different types of SMF files.

SMF type 0

SMF chunk type 0 contains a maximum of one track. This is the most simple type of SMF file. For example, this type can be used for storing the instrument tracks of a sequencer individually.

SMF type 1

The SMF type 1 is much more versatile because it can contain several tracks at once. The different tracks are always stored sequentially within the file; each track also forms an individual track-chunk.

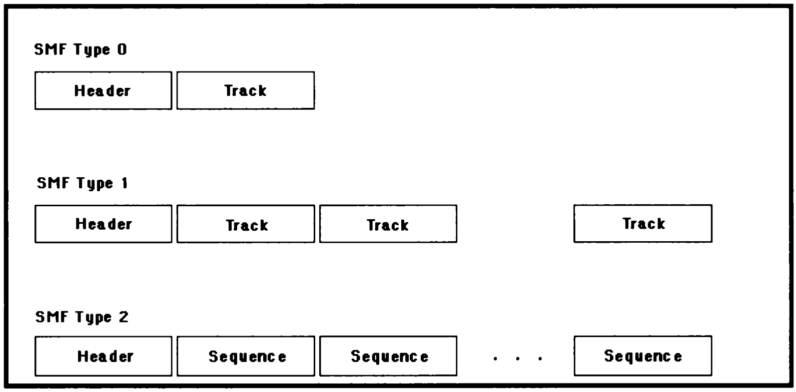
However, eventually these tracks still contain information that must be interpreted simultaneously. This means that once all tracks have been loaded, they must be processed chronologically, parallel to one another.

SMF type 2

Although this type also contains different sequences stored in tracks, the data contained within these types don't have to be played back simultaneously. Instead they can be independent pieces.

However, in this type of MIDI file, you should store only sequences that belong to a single piece of music, even though you can store anything in it.

SMF type 1 is probably the most frequently used file type.



Different SMF file types

Header chunk

The MThd chunk is always the first chunk in a SMF file. As usual, it consists of the four-byte chunk ID and the four-byte length value.

The length of the most frequently used header is 6. So this means that the length bytes are followed by 6 data bytes.

SMF Type info

The first two data bytes contain the type of the SMF file (i.e., either the value 0, 1, or 2).

Flip-flopped WORDs

When evaluating WORD values from a SMF file remember that the sequence of Lo and Hi bytes is exactly the opposite of the sequence used by Intel processors. This means that you must swap the high and low bytes before obtaining the correct value.

The same problem occurs with the 32-bit values used in the chunk lengths. You must use the same values in these bytes to obtain the correct result. As you may remember, we've already encountered this obstacle with Amiga MOD files.

Number of tracks

The next two bytes contain the number of tracks contained in the SMF files as a WORD value. For SMF files of type 0, this value is always 1.

The next two bytes must be interpreted differently, depending on whether bit 15 is set.

If bit 15 isn't set, then bits 14 - 0 represent a value that indicates how many delta time ticks are allotted to a quarter note. For

example, the bytes "00000000 01100000" specify 96 delta time ticks per quarter note.

However, when bit 15 is set to 1, this indicates that a time code was used for this MIDI file.

Time code method

The time code method is used, for example, for processing video tape. Each location of the video tape is imprinted with time information. So any desired location on the tape can be accessed.

An advantage of this process is that video tape can be cut and spliced down to fractions of a second. The time intervals used are primarily "frames per second", which can then also be specified in hours, minutes, seconds, frames, and hundredth frames.

When music videos are edited, both the video data and the accompanying music must be precisely matched. This is one of the reasons why music formats using a time code method have been developed. This method allows the video and music data to be spliced together within fractions of a second.

SMPTE (the MIDI time code method) is the preferred time code method. This method requires only a minimum amount of memory in addition to the MIDI data. Remember that MIDI data are transferred serially.

Although time codes must also follow certain standards, we won't discuss this at this time.

You must also know one more thing about the time code method in reference to SMF files. When bit 15 is set, bits 14 - 8 represent a value in the form of a negative 2's complement, from which you'll be able to tell how many frames per second the time code specifies. Bits 7 through 0 indicate the resolution of the time code signal.

This completes the description of the data found in the file header. Now let's move on to the first track.

Track chunk

The track chunk again begins with the chunk ID and the 32 bits for the length value. These are followed by a number of events, which form the actual data of the track.

Delta time

Each event is preceded by a value that indicates how much time must pass before the event is executed. This value is referred to as

delta time. The delta time bytes are a permanent syntax component of an event.

The way in which these bytes are coded is slightly unusual. To save space, the delta time value doesn't have a fixed number of length bytes. Instead, it uses bit 7 of a length byte to indicate that another length byte will follow.

This means that only bits 6 through 0 are actually used for the length value. Theoretically, this allows any number of bytes to be added to one another, forming numbers of any desired size.

However, a maximum number of 4 bytes should be used. This means that the shortest delta time sequence is a byte of the type "0xxxxxxx". It will contain a value between 0 and 127.

The longest delta time sequence would be "1xxxxxxx 1xxxxxxx 1xxxxxxx 0xxxxxxx".

If you want to represent the number 255 in delta notation, you must use more than one byte. Using a second byte, this number would look as follows: "10000001 01111111". This convention is almost identical to the high-byte/low-byte arrangement used for normal data values.

The only difference is that the value 256 cannot be used as the value of each byte for the calculation of the actual value. Instead, the number 128 must be used because the seventh bit isn't included. Later we'll show you how to do this within a program.

The delta time sequence is followed by the event's number. Three different types of events can occur within a SMF file. These are MIDI events, System Exclusive events, and Meta events.

MIDI events

Before the format definition by the IMA, MIDI events also included System Common messages. However, now these are only Channel Mode messages.

We discussed the formats and sizes of the individual Channel Mode messages in Chapter 4.

System Exclusive messages are much more interesting. Since the contents of these messages aren't determined by the MIDI standard, only the length information for this message must be added.

A System Exclusive event in an SMF file begins, as usual, with the byte $F0. This byte is followed by the length of the following data, instead of by the first data byte. This value is encoded in the same way as the delta time value.

The end-byte of a System Exclusive message is usually byte $F7. Since the length of the data has already been specified, this byte isn't really necessary. However, it has been retained for other reasons.

Within MIDI files, it's possible to divide particularly lengthy system exclusive events into more manageable packages. This enables the time code data to be inserted at the required locations.

The second portion of such a divided message begins with the byte $F7. Although the message shouldn't begin with $F0, it should always end with $F7 to indicate the actual end of the System Exclusive message.

Meta events

Meta events are the true specialty of SMF files. They can contain various kinds of data.

We'll discuss how to use these individual events in more detail later.

Event $00 (00) - Sequence number

If event 0 is included in an SMF file, it must be placed at the very beginning of a track, before the first MIDI data that must be transferred.

This event has two data bytes representing a 16-bit value containing the sequence number of the subsequent data.

This is important primarily for SMF files of type 2. The event can be used to correctly "number" sequences contained in such a file. If these events are omitted, the exact sequence found in the file is used.

Event $01 (01) - General text

This event allows you to insert any desired text into an SMF file. The ID byte $01 is followed by the length of the text. As in all other text events, this is represented in delta format. This code is followed by the text in ASCII format.

Event $02 (02) - Copyright text

This event also contains text. However, the event number 1 is reserved specifically for copyright notices. Copyright notices are usually placed in the first track of the file.

Event $03 (03) - Track name

Event 3 provides another way to store text in the SMF file. The number 3 is reserved for information on the track data or the sequence data.

You can store general titles, such as "bass intro" or "drum solo", in this event so you can easily identify the music data stored in that track.

In SMF files of type 2, you can use this event to store information on the sequence.

Event $04 (04) - Instrument name

Event 4 is reserved for the description of instrument voices. You can use this event, for example, to describe a program change event within a track or to indicate which instrument should be played on the appropriate channel after a channel prefix event (event 32).

Event $05 (05) - Song lyric

When these events were being developed, both the music and the lyrics were included in an SMF file. This means that a song text event can be inserted at the start of each syllable that must be sung.

Event $06 (06) - Marking

Event 6 can be used to mark specific spots within a sequence so these can be located easily at a later time. Again, the length value is also stored in delta format.

Event $07 (07) - Cue point

This event is useful when MIDI files are used with video material. A cue point is a spot to which you can cue or rewind. So the descriptive text in event 7 can contain information on this particular spot.

Events $08-$0F (08-15) - Unused

Events 8 through 15 are reserved for text events, although they are currently unused.

Event $20 (32) - Channel prefix

This event is used to show that all subsequent Meta events are directed at a specific MIDI channel. The channel number is stored in a single byte following the event ID. This channel selection remains active until a new channel prefix event occurs or until the next MIDI event is encountered.

This means that event 32 doesn't actually divert MIDI data to another channel. Instead, it's used only for informational purposes. Therefore, it can be used, as we mentioned above, with event 4 if the channel number of an instrument isn't already specified in event 4.

Event $2F (47) - End of track

This event indicates the end of a track. Since additional data doesn't follow, the length of this event is 0. However, due to the standardized notation of events, this length value is also stored in delta format after the event ID. An event 47 must always be placed at the end of the track.

Event $51 (81) - Set tempo

Event 51 contains, following the length value, a 24-bit number (3 bytes). This number represents the number of microseconds (millionths of a second) within one quarter note. This means that a value of 1,000,000 indicates that one quarter note must be played each second.

Event $54 (84) - SMPTE offset

Event 84 is used with the SMPTE time code method, which we discussed earlier.

This event contains five data bytes. These bytes specify the hours, minutes, seconds, frames, and hundredth frames.

Event $58 (88) - Time signature

In event 88 you'll find four data bytes in addition to the delta length value. The first data byte contains the numerator of a time signature fraction and the second byte contains the corresponding

denominator as 2 exponential. So, the value 3 in this byte actually indicates the number "2 to the 3rd power" (i.e., 8).

The third data byte specifies how many MIDI clocks must represent one metronome click and the fourth byte contains the number of 32ndth notes that correspond to one quarter note (24 MIDI clocks).

The setting for a 4/4 time signature with a metronome click at each quarter note and eight 32ndth notes per 24 MIDI clocks would look as follows: FF 58 04 04 02 18 08.

Event $7F (127) - Sequencer

The SMF standard contains event 127 primarily to accommodate the different sequencers. This event is very similar to system exclusive messages, since they can be freely adapted to various hardware or software requirements by sequencer manufacturers.

So we cannot provide an example of a sequencer specific event at this point. The interpretation method for the event's data bytes depends on the individual manufacturer.

Now you can examine a MIDI file more closely. The following program should help you do this.

5.5.2 The MIDSCRIPT program

For this program, we purposely used a name that's similar to MODSCRIPT. Both programs perform the same task: Creating script files from existing music files.

MODSCRIPT is capable of reading Amiga Soundtracker modules and MIDSCRIPT can be used for standard MIDI files with the .MID filename extension. The program loads a MIDI file into memory and evaluates its contents.

The program is written in Turbo Pascal. So we had to deal with some complicated arithmetic for the program's pointers. However, if you're a C programmer and you wish to adapt this program for C, you shouldn't have any problems with pointer arithmetic because it's common in the C language.

Therefore, you should be able to convert the structure of this program to C easily. To increment a pointer, for example, beyond a segment limit, requires some complicated maneuvers, if a pointer of the type "huge" is used.

MIDTOOL

The MIDTOOL.PAS unit contains all the procedures that are needed for evaluating a SMF file and creating a corresponding ASCII text file. Several global variables, constants, and data types must be used.

The constants MChunkH and MChunkT contain the identifiers for the existing data types MThd and MTrk (header and track) in the form of character strings.

Then several data types, which are helpful for the general operation with SMF files, are declared.

ChunkName

The ChunkName type is an array with four elements of ASCII characters. It's used for variables that must store the name of a chunk.

ChunkSize

The ChunkSize type is also an array with four elements. However, these elements are of the type byte. This array is used to store the 32-bit length value of a chunk.

If a 4-byte data type, such as LongInt, is used instead of this array, you'll receive an incorrect value because the swapped byte sequence must first be interpreted correctly.

DeltaType

The type DeltaType is a record that also contains a four-element byte array. This type was created for variables that contain the flexible length code value of SMF files.

So the record structure also contains the Count element. This element contains the number of entries that are actually used so the procedure performing the calculation knows at which byte to begin.

Although this value could be omitted by instead saving the delta bytes in reverse order from the end of their array, this method is easier to understand.

MIDChunkInfo

The file type MIDChunkInfo is used mainly for typecase access to your PC's memory. As long as array structures remain reasonably small, they can still be swapped over the stack. Therefore, this typecasting can be used to access the chunk ID and chunk length bytes directly and return these to program procedures as function values.

MIDHeader-Chunk The type MIDHeaderChunk has been created specifically for the evaluation of header data. The header size has remained unchanged (6 bytes) in the form of three WORD values.

This data type allows you to access these first three word values through typecasting. It also provides the chunk ID and length, which are also available through the type MIDChunkInfo.

Then the global variables, which can also be accessed outside of the unit, are declared.

MIDErrStat The first of these is the unit's error status variable, which is called MIDErrStatus, as in the previous examples.

The PrintMIDErrMessage procedure can be used to display error messages on the screen.

Since the palette of functions in this unit isn't as extensive as in previous units, this unit doesn't require as many error messages. Only the 2xx and 3xx error message numbers are included in this unit.

Error 200 Error number 200 indicates that the specified MIDI file couldn't be found.

Error 210 Error number 210 is activated when available memory cannot be allocated to the MIDI file. This may occur when the program that's using the MIDTOOL unit hasn't set an upper memory limit with the {$M} compiler directive. In this case, your program would reserve all available memory and wouldn't leave any room for further MIDI data.

Error 220 Error 220 indicates that the loaded file isn't a standard MIDI file. This is the case if the file doesn't begin with a header chunk.

Error 300 This memory error occurs only if you've tried to free a memory area that hasn't been allocated.

Global variables The following global variables can be evaluated only after an SMF file has been loaded successfully.

MIDFormat The MIDFormat variable then contains the type number of the loaded SMF file. As we mentioned, this number can be either 0, 1, or 2.

MIDTracks

The MIDTracks variable will contain the number of tracks contained in the loaded file. For SMF files of type 0, this is always 1.

MIDDivis

After the file has been loaded successfully, the MIDDivis variable will contain the value of the file's division bytes, which we mentioned.

Functions

The functions and procedures of this unit have been tailored to the requirements of the MIDBuildScript function. However, individual functions can also be used independently, and others can even be modified for your own use.

The main goal of these functions is to demonstrate how you can access the data contained in SMF files.

This is also why we used Turbo Pascal. The pointer operations and typecasting are much more complex procedures than in C.

The main function

The primary portion of the unit centers around the MIDBuildScript function.

Function MIDBuildScript

Only two parameters must be passed to this function. These parameters are the name of the desired MIDI file and a flag that specifies whether the data should be displayed on the screen or written to a text file.

Because of the many MIDI events contained in a MIDI file, always ensure there is enough disk space to create the text file. With longer songs, these events can easily contain thousands of bytes.

At the start of the function, the flag that was specified at the DOS command line is checked. If this value is 1, a text file is created on disk; if it's 0, the screen is opened as an output "file".

Then the specified file is loaded into memory. If an error occurs, the function is aborted and the appropriate error number is assigned to MIDErrStat.

If the file has been loaded successfully, the title for the text data, which also contains the file's name and size in bytes, is displayed.

Then the internal global variable MIDGlobSize is set to 0. This variable is used to check whether the evaluation has reached the end of the loaded file.

Since the data are already located in memory, they don't include an EOF message (End Of File).

Then the pointer to these data is copied from the variable MIDBuffer to the variable MIDIntern. The address contained in MIDBuffer remains the same throughout the entire program, while MIDIntern continually follows the evaluation of the data.

The subsequent Repeat/Until..... loop reads data from the SMF file until the variable MIDGlobSize is as large as the size of the loaded file. This indicates that all data contained in this file have been processed.

Then the memory occupied by the data is freed again and the opened text file is closed. This completes the tasks of MIDBuildScript.

Procedure MIDInterpretChunk

This procedure decodes a chunk once it has been found.

First the chunk ID is copied to the ActChunk variable and the chunk length is copied to the ActSize variable.

Header chunk

If the value in ActChunk has identified the chunk as a header chunk, the header type and size are displayed first. This is followed by the actual information contained in the header.

Then the chunk must be skipped because its bytes aren't read by increasing the pointer sequentially. Instead, the bytes are assigned to variables when the file is loaded. The chunk is skipped by the MIDSkipChunk procedure.

Track chunk

If the chunk is recognized as a track chunk, several additional tasks must be performed.

As with the header, the chunk type and size in bytes are first displayed. Also, the bytes occupied by the chunk ID and length are skipped with MIDIncrementPtr.

Before you can decode the chunk data, another task must be performed. The internal global variable MIDLoclSize must be set to 0. This variable is used to check whether all the data bytes of the track have been processed.

The procedure will evaluate data bytes until the value of this variable reaches the size of the track's length byte.

To evaluate event syntax, first delta time must be designated. So the delta bytes are first read by the MIDReadDeltaValue procedure. The two variables MIDBuffer and DeltaTime are passed as function parameters. MIDBuffer contains the pointer out of MIDIntern from MIDBuildScript. At this time, it points to the start of this track's first event.

The variable DeltaTime is of the type DeltaType. The delta bytes obtained in the MIDReadDeltaValue procedure, and the information on the number of bytes that are relevant, are stored in this variable. With these values, the MIDCalcDeltaValue procedure is called. This procedure returns the actual value of the delta bytes.

Then the next byte is read into the ActData variable. This byte contains the number of events that will now follow.

MIDI events

This number indicates the type of event with which you're working. If it's a number between $80 and $EF, the event is a MIDI event. In this case, the program jumps to the MIDScanMIDIEvent procedure.

Meta events

If the byte contains $FF, the event is a Meta event, in which case the program calls the MIDScanMetaEvent procedure.

SysEx events

If the value of this byte is $F0, the event is a System Exclusive event. Such events are evaluated by the MIDScanSysEvent procedure.

If the byte's value doesn't match any of the numbers representing MIDI events, Meta events or SysEx events, the subsequent data is simply ignored and the reading continues at the next chunk.

Once all data bytes have been processed, the MIDInterpretChunk procedure is completed.

Procedure MIDScanMIDIEvent

This procedure is called when an event has been recognized as a MIDI event. The procedure checks the numbers of the MIDI event and the corresponding messages are displayed.

An SMF file can contain only channel mode messages. As you may remember from Chapter 4, the seventh bit of all MIDI commands is set to 1. This means that their value is always more than 128.

A MIDI event always consists of 16 different numbers—one for each MIDI channel. This is taken into consideration in the MIDScanMIDIEvent procedure, which evaluates the individual channel mode messages according to the MIDI standard. So the number of data bytes is also known.

Procedure MIDScanMetaEvent

This procedure is called when an event is recognized as a Meta event.

We've already described Meta events in detail. The length of Meta events appears, as usual, in the familiar delta format. This length value is determined and assigned to the MetaSize variable.

Depending on the individual event number, the procedure displays the information contained in the event. If an unrecognized number is encountered, the event is simply skipped and the message "Event Unknown" is displayed.

Procedure MIDScanSysExEvent

This procedure is quite minimal, since simply decoding several System Exclusive messages isn't very useful. These messages differ depending on the manufacturer and on the computer.

So this procedure simply displays the message "System Exclusive" as well as length of the message in bytes. The rest of the message is skipped.

Almost universal

The functions we've listed so far have all been program sections designed to display text on the screen. They must be modified to perform other tasks.

However, the following functions can be used in various ways, since they perform tasks that must be performed in the background to help the functions listed above.

Function MIDGetBuffer

The MIDGetBuffer function is used to load a MIDI file into memory.

The structure of this function is similar to the structures of the load functions in VOCTOOL and CMFTOOL.

The parameters required by this function are a pointer variable that points to the data in memory, once these have been

successfully loaded, and the name of the desired file as a character string.

The returned function value will either be TRUE or FALSE, depending on whether an error has occurred. In case of an error, the error number 200 through 220 may apply.

If the file was loaded successfully, the most important data of the header is copied to the global variables described above.

Function MIDFreeBuffer

This function frees the memory that's occupied by the loaded MIDI data. As usual, the only possible error that can occur during this function call is number 300.

The only required parameter is the pointer identifying the MIDI data.

Function MIDCalcSize

This function is used to transform the 32-bit length values, found in chunks, into a proper LongInt number.

The function requires only the 4-byte array, which you've read from the SMF file, as its parameter. Then it returns the correct size value. To do this, the function reverses the bytes' positions and multiplies these by factors of 256 before adding them.

So the procedure compensates for the reversed byte convention used by Intel chips.

Procedure MIDReadDeltaValue

This function automatically reads delta time values from an SMF file.

The only parameters needed for this function are the pointer to the current position in memory and an empty variable of the type DeltaType.

The procedure then reads as many bytes as the current delta value contains. For this, the function checks whether bit 7 of the current byte is set, which indicates that another byte will follow.

The maximum length of these values is four bytes. However, the actual number of bytes is returned to the delta value variable through the structure element Count.

The element Data then contains the corresponding bytes.

Function MIDCalcDeltaValue

This procedure can be used to transform the bytes, which were read by the previous function, into a usable length value.

Simply specify the variable, to which the function value of MIDReadDeltaValue was just assigned, as the function parameter. You'll receive the actual delta size as a LongInt value.

To receive this value, the function multiplies the values of these bytes by factors of 128; bit 7 is ignored, since it doesn't belong to the actual delta value.

Procedure MIDSkipChunk

The MIDSkipChunk procedure is useful when unrecognizable chunks must be skipped.

Currently only the chunk types header and track are being used. However, according to the IFF format and the SMF format, it should also be possible to search unknown files for recognizable data.

Therefore, the length values specified with each chunk makes it possible to skip the chunk if its data is unusable.

This is exactly the procedure used in MIDSkipChunk. It reads the length of the current chunk and adds 8 bytes to this value to account for the chunk ID and the chunk length bytes. Then the current pointer is incremented by exactly the same number of bytes as indicated by this chunk length.

This procedure is used within MIDInterpretChunk whenever an unrecognizable chunk is encountered.

However, this procedure is also used during the evaluation of the header chunk. Although header chunks currently contain 6 bytes, this may change in the future.

Therefore, instead of skipping these fixed 6 bytes, the entire chunk is skipped following the typecast evaluation. This is possible because the current pointer isn't incremented by the typecast operation and is therefore still pointing to the start of the header even after this operation.

Procedure MIDIncrementPtr

The most frequently used procedure of this unit is MIDIncrementPtr. This procedure is used each time the pointer for the MIDI data must be incremented by a certain value.

The current pointer and the number of bytes, by which the pointer must be incremented, are passed to the function as parameters. This procedure helps implement the necessary pointer arithmetic in Pascal.

The procedure checks whether a segment limit has been passed while the pointer was incremented. If this occurred, the segment is also incremented accordingly.

Several segments

Although the procedure doesn't permit negative values (i.e., decrementing the pointer), the procedure is flexible enough that the pointer can be incremented over several segment limits.

In a MIDI file, this probably won't be the case even when several chunks are skipped at once. However, you may want to use this procedure in other programs for examining larger amounts of data in memory.

For such applications, the procedure can be rewritten so the values of MIDGlobSize and MIDLocSize are also incremented automatically.

Incrementing these two global variables ensures that they will always contain the correct value for the position within the entire file (MIDGlobSize) as well as within a single chunk (MIDLocSize), on the level of the entire unit.

So MIDGlobSize must be set to 0 when the file is loaded, and MIDLocSize must be set to 0 each time a chunk is completed, or at least before the evaluation of the next chunk begins.

The MIDInterpretChunk and MIDBuildScript procedures both consider this, and rely heavily on both of these variables.

Ideas for expansions

Modifications

This completes the description of this unit's functions. By using these functions and the information presented here, you can create much more complex and flexible programs.

For example, you can not only create a simple script file from an SMF file, but also evaluate the values of each MIDI event so the program automatically generates sheet music from the SMF file by converting the note values in the file to notes on the music staff.

You'll probably think of many uses for the data in an SMF file. Your imagination is the only limitation.

```
UNIT MIDTool;
{
 *********************************************************************
 *      Unit for reading information from an SMF file               *
 *********************************************************************
 *                      (C) 1992 Abacus                              *
 *                     Author : Axel Stolz                           *
 *********************************************************************
}

INTERFACE

Uses Dos, Crt;

CONST
   MIDToolVersion = 'v1.0';

   MChunkH = 'MThd'; { Header chunk ID for SMF files }
   MChunkT = 'MTrk'; { Track chunk ID for SMF files  }

TYPE
   ChunkName = ARRAY[0..3] OF CHAR; { Array for chunk ID           }
   ChunkSize = ARRAY[0..3] OF BYTE; { Array for 32-bit chunk size  }
   DeltaType = RECORD
                Data : ARRAY[0..3] OF BYTE; { DeltaTime data array }
                Count: BYTE;
                END;

{ Data type for all necessary information about a chunk }
{ Typecast access to chunk ID and chunk size            }
   MIDChunkInfo = RECORD
              ckID   : ChunkName;
              ckSize : ChunkSize;
              END;

{ Data type for all necessary data about the        }
{ header chunk of an SFM file for typecast accesses }
   MIDChunkHeader = RECORD
              ckID   : ChunkName;
              ckSize : ChunkSize;
              Format : WORD;
              Tracks : WORD;
              Divis  : WORD;
              END;
{ MIDFiletype is the standard file type }
```

```
   MIDFileType = FILE;

VAR
   MIDErrStat  : WORD;      { Global variable for error status      }
   MIDFormat   : WORD;      { Global variable for MIDI file format  }
   MIDTracks   : WORD;      { Global variable for number of tracks  }
   MIDDivis    : WORD;      { Global variable for division info     }

PROCEDURE PrintMIDErrMessage;
FUNCTION  MIDGetBuffer(VAR MIDBuffer:Pointer;
                           MIDFilename:String):BOOLEAN;
FUNCTION  MIDFreeBuffer (VAR MIDBuffer : Pointer):BOOLEAN;
FUNCTION  MIDCalcSize(MSize : ChunkSize): LongInt;
FUNCTION  MIDCalcDeltaValue(DeltaVal:DeltaType):LongInt;
PROCEDURE MIDIncrementPtr(VAR MIDBuffer : Pointer;
                              InternSize : LongInt);
PROCEDURE MIDSkipChunk(VAR MIDBuffer : Pointer);
PROCEDURE MIDReadDeltaValue(VAR MIDBuffer:Pointer;
                            VAR DVal:DeltaType);
PROCEDURE MIDScanMetaEvent(VAR MIDBuffer : Pointer);
PROCEDURE MIDScanMIDIEvent(VAR MIDBuffer : Pointer);
PROCEDURE MIDInterpretChunk(VAR MIDBuffer : Pointer);
FUNCTION  MIDBuildScript(MIDFilename : String; Flag : BYTE):BOOLEAN;

IMPLEMENTATION
VAR
   Regs        : Registers;
   MIDFileSize : LongInt;   { Global variable for file size }
   MIDGlobSize : LongInt;   { Global variable for global size }
   MIDLoclSize : LongInt;   { Global variable for chunk size }
   M           : Text;      { Global variable for MID text file }

PROCEDURE PrintMIDErrMessage;
{
 * INPUT    : None
 * OUTPUT   : None
 * PURPOSE  : Displays MID error text on the
 *            screen without changing error status.
}
BEGIN
   CASE MIDErrStat OF
      200 : Write(' MID file not found ');
      210 : Write(' No memory free for MID file ');
      220 : Write(' File is not in MID format ');
      300 : Write(' Memory allocation error occurred ');
      END;
   END;

FUNCTION Exists (Filename : STRING):BOOLEAN;
{
 * INPUT    : Filename as string
 * OUTPUT   : TRUE, if file exists, otherwise FALSE
 * PURPOSE  : Checks whether a file already exists,
 *            and returns a Boolean expression.
```

```
}
VAR
   F : File;
BEGIN
   Assign(F,Filename);
{$I-}
   Reset(F);
   Close(F);
{$I+}
   Exists := (IoResult = 0) AND (Filename <> '');
   END;

PROCEDURE AllocateMem (VAR Pt : Pointer; Size : LongInt);
{
 * INPUT    : Buffer variable as pointer, buffer size as LongInt
 * OUTPUT   : Pointer to buffer in variable or NIL
 * PURPOSE  : Reserves as many bytes as specified by size and then
 *            places the pointer in the Pt variable. If not enough
 *            memory is available, then Pt points to NIL.
}
VAR
   SizeIntern : WORD; { Size of buffer for internal calculation }
BEGIN
   Inc(Size,15);                 { Increase buffer size by 15 ...  }
   SizeIntern := (Size shr 4);   { and then divide by 16.          }
   Regs.AH := $48;               { Load MS-DOS function $48 in AH  }
   Regs.BX := SizeIntern;        { Load internal size in BX        }
   MsDos(Regs);                  { Reserve memory                  }
   IF (Regs.BX <> SizeIntern) THEN Pt := NIL
   ELSE Pt := Ptr(Regs.AX,0);
   END;

FUNCTION  MIDGetBuffer(VAR MIDBuffer:Pointer;
                       MIDFilename:String):BOOLEAN;
{
 * INPUT    : Variable for buffer as pointer, filename as string
 * OUTPUT   : Pointer to buffer with MID data, TRUE/FALSE
 * PURPOSE  : Loads a file into memory and returns a value of TRUE
 *            if successfully loaded, otherwise returns FALSE.
}
VAR
   FPresent   : BOOLEAN;
   MIDFile    : MIDFileType;
   Segs       : WORD;
   Read       : WORD;

BEGIN
   FPresent := Exists(MIDFilename);

{ MID file could not be found }
   IF Not(FPresent) THEN BEGIN
      MIDGetBuffer := FALSE;
      MIDErrStat   := 200;
      EXIT
```

```
      END;

   Assign(MIDFile,MIDFilename);
   Reset(MIDFile,1);
   MIDFileSize := Filesize(MIDFile);
   AllocateMem(MIDBuffer, MIDFileSize);

{ Not enough memory for the MID file }
   IF (MIDBuffer = NIL) THEN BEGIN
      Close(MIDFile);
      MIDGetBuffer := FALSE;
      MIDErrStat   := 210;
      EXIT;
      END;

   Segs := 0;
   REPEAT
      Blockread(MIDFile,Ptr(Seg(MIDBuffer^)+4096*Segs,
                Ofs(MIDBuffer^))^,$FFFF,Read);
      Inc(Segs);
      UNTIL Read = 0;
   Close(MIDFile);

{ File is not in MID format }
   IF ( MIDChunkInfo(MIDBuffer^).ckID <> MChunkH) THEN BEGIN
      MIDGetBuffer := FALSE;
      MIDErrStat := 220;
      EXIT;
      END;

{ Loading successful }
   MIDGetBuffer := TRUE;
   MIDErrStat   := 0;

{ Read MIDI file type in global variable }
   MIDFormat := Swap(MIDChunkHeader(MIDBuffer^).Format);

{ Read number of tracks contained in global variable }
   MIDTracks := Swap(MIDChunkHeader(MIDBuffer^).Tracks);

{ Read division value in global variable }
   MIDDivis  := Swap(MIDChunkHeader(MIDBuffer^).Divis);
   END;

FUNCTION MIDFreeBuffer (VAR MIDBuffer : Pointer):BOOLEAN;
{
 * INPUT    : Pointer to buffer as pointer
 * OUTPUT   : None
 * PURPOSE  : Releases memory allocated by the MID data.
}
BEGIN
   Regs.AH := $49;              { Load MS-DOS function $49 in AH    }
   Regs.ES := seg(MIDBuffer^);  { Load segment of memory in ES      }
   MsDos(Regs);                 { Release memory again              }
```

```
   MIDFreeBuffer := TRUE;
   IF (Regs.AX = 7) OR (Regs.AX = 9) THEN BEGIN
      MIDFreeBuffer := FALSE;
      MIDErrStat := 300         { A DOS error occurred during       }
      END;                      { release.                          }
   END;

FUNCTION MIDCalcSize(MSize : ChunkSize): LongInt;
{
 * INPUT    : 32 bit number as 4 byte array
 * OUTPUT   : Real 32 bit value as LongInt
 * PURPOSE  : Converts the reflected 32-bit value of the MID file
 *            to a real 32-bit LongInt value.
}
VAR
   Power : REAL;
   Dummy : LongInt;
   Count : BYTE;

  BEGIN
   Dummy := 0;
   FOR Count := 3 DOWNTO 0 DO BEGIN
    Power := Exp(Count * Ln(256));
    Dummy := Dummy + (Trunc(Power)*MSize[3-Count]);
    END;
   MIDCalcSize := Dummy;
   END;

FUNCTION  MIDCalcDeltaValue(DeltaVal:DeltaType):LongInt;
{
 * INPUT    : DeltaVal as a record of DeltaType
 * OUTPUT   : Real Delta value as LongInt
 * PURPOSE  : Calculates the real value from the scanned Delta byte
 *            by masking the highest bit and then raising the values
 *            to corresponding higher values. This routine is used
 *            for determining Meta event sizes and DeltaTimes.
}
VAR
   Power : REAL;
   Dummy : LongInt;
   Loop  : BYTE;

BEGIN
   Dummy := 0;
   WITH DeltaVal DO BEGIN
      FOR Loop := (Count-1) DOWNTO 0 DO BEGIN
       Power := Exp(Loop * Ln(128));
       Dummy := Dummy+(Trunc(Power)*(Data[(Count-1)-Loop] AND $7F));
       END;
      END;
   MIDCalcDeltaValue := Dummy;
   END;

PROCEDURE MIDIncrementPtr(VAR MIDBuffer : Pointer;
```

```
                              InternSize : LongInt);
{
 * INPUT    : Pointer variable to current position in the MID data
 *            as reference parameter, increment value as LongInt
 * OUTPUT   : None (new pointer from reference)
 * PURPOSE  : Increments the value of the passed pointer by the value
 *            InternSize. Increases beyond a segment limit are taken
 *            into account.
}
VAR
  Segment    : WORD;
  Offset     : WORD;
  Offnew     : WORD;
  SegCount   : LongInt;

BEGIN
{ Negative increment not allowed }
  IF (InternSize < 0) THEN Exit;

  Segment := Seg(MIDBuffer^); { Determine current segment }
  Offset  := Ofs(MIDBuffer^); { Determine current offset  }

{ How many segments must be incremented? }
  SegCount := (InternSize DIV $10000);

{ Calculate new offset address }
  Offnew  := Offset+InternSize;

{ Was the increment value smaller than a segment, but the results  }
{ still exceed the segment limit? Increment segment by 1.          }
  IF ((Offnew <= Offset) AND (SegCount = 0) AND (InternSize > 0))
  THEN SegCount := 1;

  INC(Segment, SegCount*$1000);

  MIDBuffer := Ptr(Segment,Offnew); { Reassemble pointer   }
  INC(MIDGlobSize,InternSize); { Adapt global variable also }
  INC(MIDLoclSize,InternSize); { Adapt global variable also }
  END;

PROCEDURE MIDSkipChunk(VAR MIDBuffer : Pointer);
{
 * INPUT    : Pointer variable to current position in the MID data as
 *            reference parameter
 * OUTPUT   : None (new pointer from reference)
 * PURPOSE  : Skips the adjacent chunk with the help of the size
 *            information in the chunk itself.
}
VAR
  InternSize : LongInt;
  Segment    : WORD;
  Offset     : WORD;

BEGIN
```

```
{ Determine the size of the data, and then add the 8 bytes for }
{ the ID bytes and the size bytes                               }
  InternSize := MIDCalcSize(MIDChunkInfo(MIDBuffer^).ckSize)+8;
  MIDIncrementPtr(MIDBuffer,InternSize);
  END;

PROCEDURE MIDReadDeltaValue(VAR MIDBuffer:Pointer; VAR DVal:DeltaType);
{
 * INPUT    : Pointer variable to current position in MID data as
 *            reference parameter, record for DeltaTime as reference
 * OUTPUT   : (new pointer from reference)
 *            (filled array in record from reference)
 *            (number of scanned Delta bytes in reference record)
 * PURPOSE  : Selects the size coded Delta values from the MID data
 *            and writes them to the passed array of the record.
}
VAR
   ActData  : BYTE;
   ActDelta : BYTE;
BEGIN
{ Delete passed array, so that there are no longer any "false" }
{ bytes contained. This is only a cosmetic improvement.        }
   FOR ActDelta := 0 to 3 DO DVal.Data[ActDelta] := 0;
{ Set bit 7 }
   ActData  := 128;
   ActDelta := 0;
{ while bit 7 is set, read Delta bytes }
   WHILE ((ActData AND $80)=$80) DO BEGIN
      ActData := BYTE(MIDBuffer^);
      DVal.Data[ActDelta] := ActData;
      INC(ActDelta);
      MIDIncrementPtr(MIDBuffer,1);
      END;
   DVal.Count := ActDelta;
   END;

PROCEDURE MIDScanSysExEvent(VAR MIDBuffer : Pointer);
{
 * INPUT    : Pointer variable to current position in MID data
 *            as reference parameter
 * OUTPUT   : None (new pointer from reference)
 * PURPOSE  : If the passed pointer points to a SysEx event in the
 *            MID data in memory, then the message is displayed here,
 *            and the rest of the events are skipped.

}
VAR
   DeltaTime : DeltaType;
   SysExSize : LongInt;
BEGIN
   WriteLn(M,'System Exclusive (');
   MIDIncrementPtr(MIDBuffer,1);
{ Determine size of the Meta event }
   MIDReadDeltaValue(MIDBuffer, DeltaTime);
```

```
   SysExSize := MIDCalcDeltaValue(DeltaTime);
   WriteLn(SysExSize,' Bytes)');
   MIDIncrementPtr(MIDBuffer, SysExSize);
   END;

PROCEDURE MIDScanMetaEvent(VAR MIDBuffer : Pointer);
{
 * INPUT    : Pointer variable to current position in the MID data
 *            as reference parameter
 * OUTPUT   : None (new pointer from reference)
 * PURPOSE  : If pointer passed points to a Meta event in the MID data
 *            in memory, then the event is interpreted according to
 *            its data, and the appropriate text is displayed.
}
TYPE
  Overhead = ARRAY[0..5] OF BYTE; { Type for necessary Typecast }
VAR
   ActEvent : BYTE;      { Contains the ID 'FF' for Meta events     }
   EventType: BYTE;      { Contains number of current Meta event    }
   DeltaVal: DeltaType;
   MetaSize : LongInt;
   MetaWord : WORD;
   MetaLong : LongInt;
   ActCount : LongInt;

BEGIN
   Write(M,'Meta event : ');
   ActEvent  := Overhead(MIDBuffer^)[0]; { Read ID             }
   EventType := Overhead(MIDBuffer^)[1]; { Read number of event }
   MIDIncrementPtr(MIDBuffer,2);
{ Determine size of Meta event }
   MIDReadDeltaValue(MIDBuffer, DeltaVal);
   MetaSize := MIDCalcDeltaValue(DeltaVal);
   CASE EventType OF
{ Event 0 - Sequence number }
   00 : BEGIN
         MetaWord := Swap(WORD(MIDBuffer^));
         WriteLn(M,'Sequence number : ',MetaWord);
         MIDIncrementPtr(MIDBuffer, MetaSize);
         END;
{ Event 1 - General text }
   01 : BEGIN
         Write(M,'Text : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Event 2 - Copyright text }
   02 : BEGIN
         Write(M,'Copyright : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
```

```
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Event 3 - Track name as text }
   03 : BEGIN
         Write(M,'Track Name : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Event 4 - Instrument name as text }
   04 : BEGIN
         Write(M,'Instrument : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Event 5 - Song text }
   05 : BEGIN
         Write(M,'Lyric : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Event 6 - Marker value }
   06 : BEGIN
         Write(M,'Marker : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Event 7 - Cue Point for video / film }
   07 : BEGIN
         Write(M,'Cue Point : ');
         FOR ActCount := 0 TO (MetaSize-1) DO BEGIN
            Write(M,CHAR(MIDBuffer^));
            MIDIncrementPtr(MIDBuffer, 1);
            END;
         WriteLn(M);
         END;
{ Events 8-15 - not yet allocated, but reserved }
   08, 09, 10, 11, 12, 13, 14, 15
       : BEGIN
          WriteLn(M,'Reserved but unallocated.');
          MIDIncrementPtr(MIDBuffer, MetaSize);
```

```
          END;
{ Event 33 - Channel selector ID }
   31   : BEGIN
          Write(M,'Channel Prefix Data ');
          WriteLn(M,BYTE(MIDBuffer^));
          MIDIncrementPtr(MIDBuffer, 1);
          END;
{ Event 47 - Display end of track }
   47   : BEGIN
          WriteLn(M,'End of Track');
          END;
{ Event 81 - Microsecond per quarter note for MIDI clock }
   81   : BEGIN
          Write(M,'Set Tempo ');
          MetaLong := 65536 * Overhead(MIDBuffer^)[0];
          INC(MetaLong,256 * Overhead(MIDBuffer^)[1]);
          INC(MetaLong,Overhead(MIDBuffer^)[2]);
          WriteLn(M,MetaLong,' Microsecs. per quarter note');
          MIDIncrementPtr(MIDBuffer, 3);
          END;
{ Event 84 - SMPTE parameters }
   84   : BEGIN
          Write(M,'SMPTE Offset ');
          Write(M,Overhead(MIDBuffer^)[0],'hr ');
          Write(M,Overhead(MIDBuffer^)[1],'min ');
          Write(M,Overhead(MIDBuffer^)[2],'sec ');
          Write(M,Overhead(MIDBuffer^)[3],'Frames ');
          WriteLn(M,Overhead(MIDBuffer^)[4],' 1/100 frames');
          MIDIncrementPtr(MIDBuffer, 5);
          END;
{ Event 88 - Parameters for MIDI clock }
   88   : BEGIN
          WriteLn(M,'Time Signature ');
          WriteLn(M,'            Numerator   : ',Overhead(MIDBuffer^)[0]);
          WriteLn(M,'            Denominator : ',Overhead(MIDBuffer^)[1]);
          WriteLn(M,'            MIDI Clocks : ',Overhead(MIDBuffer^)[2]);
          WriteLn(M,'            32/4   : ',Overhead(MIDBuffer^)[3]);
          MIDIncrementPtr(MIDBuffer, 4);
          END;
{ Event 127 - Parameters for MIDI clock }
  127   : BEGIN
          Write(M,'Sequencer Specific - ');
          WriteLn(M,'Format Unknown');
          MIDIncrementPtr(MIDBuffer, MetaSize);
          END
{ Event unknown - will not be interpreted }
   ELSE BEGIN
       WriteLn(M,'Event Unknown');
       MIDIncrementPtr(MIDBuffer, MetaSize);
       END;
      END;
   END;

PROCEDURE MIDScanMIDIEvent(VAR MIDBuffer : Pointer);
```

```
{
 * INPUT    : Pointer variable to current position in the MID data
 *            as reference parameter
 * OUTPUT   : None (new pointer from reference)
 * PURPOSE  : If pointer passed points to MIDI event in the MID data
 *            in memory, then the event is interpreted according to
 *            its data and an appropriate text is displayed.
}
TYPE
   Overhead = ARRAY[0..5] OF BYTE; { Type for necessary typecast }
VAR
   ActEvent : BYTE;  { Contains number of current MIDI event }
   ActData  : BYTE;
BEGIN
   Write(M,'(MIDI) ');
   ActEvent := Overhead(MIDBuffer^)[0];
{ Note Off event for all 16 channels }
   IF (ActEvent IN [128..128+15]) THEN BEGIN
      Write(M,'Note Off       : ');
      Write(M,'Pitch = ',Overhead(MIDBuffer^)[1]:3,',  ');
      WriteLn(M,'Volume = ',Overhead(MIDBuffer^)[2]);
      MIDIncrementPtr(MIDBuffer,3);
      Exit;
      END;
{ Note On event for all 16 channels }
   IF (ActEvent IN [144..144+15]) THEN BEGIN
      Write(M,'Note On        : ');
      Write(M,'Pitch = ',Overhead(MIDBuffer^)[1]:3,',  ');
      WriteLn(M,'Volume = ',Overhead(MIDBuffer^)[2]);
      MIDIncrementPtr(MIDBuffer,3);
      Exit;
      END;
{ Aftertouch event for all 16 channels }
   IF (ActEvent IN [160..160+15]) THEN BEGIN
      Write(M,'Aftertouch     : ');
      Write(M,'Pitch = ',Overhead(MIDBuffer^)[1]:3,',  ');
      WriteLn(M,'Volume ',Overhead(MIDBuffer^)[2]);
      MIDIncrementPtr(MIDBuffer,3);
      Exit;
      END;
{ Control change event for all 16 channels }
   IF (ActEvent IN [176..176+15]) THEN BEGIN
      Write(M,'Control change : ');
      Write(M,'Number ',Overhead(MIDBuffer^)[1]:3,',  ');
      WriteLn(M,'Value   ',Overhead(MIDBuffer^)[2]);
      MIDIncrementPtr(MIDBuffer,3);
      Exit;
      END;
{ Program change event for all 16 channels }
   IF (ActEvent IN [192..192+15]) THEN BEGIN
      Write(M,'Program change : ');
      WriteLn(M,'Number ',Overhead(MIDBuffer^)[1]:3);
      MIDIncrementPtr(MIDBuffer,2);
      Exit;
```

```
      END;
{ Aftertouch event for all 16 channels }
   IF (ActEvent IN [208..208+15]) THEN BEGIN
      Write(M,'Aftertouch     : ');
      WriteLn(M,'Volume ',Overhead(MIDBuffer^)[1]);
      MIDIncrementPtr(MIDBuffer,2);
      Exit;
      END;
{ Pitch bend event for all 16 channels }
   IF (ActEvent IN [224..224+15]) THEN BEGIN
      Write(M,'Pitch Bender   : ');
      Write(M,'Low Byte ',Overhead(MIDBuffer^)[1]:3,',  ');
      WriteLn(M,'High Byte ',Overhead(MIDBuffer^)[2]);
      MIDIncrementPtr(MIDBuffer,3);
      Exit;
      END;
   END;

PROCEDURE MIDInterpretChunk(VAR MIDBuffer : Pointer);
{
 * INPUT    : Pointer variable to current position in the MID data
 *            as Reference parameter
 * OUTPUT   : None (new pointer from reference)
 * PURPOSE  : Scans chunk data and checks for known chunk IDs. When a
 *            header chunk is found, header data is displayed. When a
 *            track chunk is found, the data are interpreted by the
 *            function above for the different events.
}
VAR
   ActSize  : LongInt;
   ActChunk : Chunkname;
   ActData  : BYTE;
   ActTime  : LongInt;
   DeltaVal : DeltaType;
   SizeCount: LongInt;
   SaveStart: Pointer;

BEGIN
   SizeCount:= 0;
   ActData  := 0;

{ Determine the type of the chunk }
   ActChunk := MIDChunkInfo(MIDBuffer^).ckID;

{ Determine the size of the chunk }
   ActSize  := MIDCalcSize(MIDChunkInfo(MIDBuffer^).ckSize);

{ Header chunk has been found }
   IF (ActChunk = MChunkH) THEN BEGIN
      WriteLn(M,'=======================================================');
      WriteLn(M,'MIDI header chunk');
      Write(M,'Chunk type     : ',(MIDChunkInfo(MIDBuffer^).ckID));
      WriteLn(M,'(',ActSize,' Bytes)');
      WriteLn(M,'MIDI file type : ', MIDFormat:1);
```

```
      WriteLn(M,'Track number  : ', MIDTracks);
      MIDSkipChunk(MIDBuffer);
      END
   ELSE
{ Track chunk has been found }
   IF (ActChunk = MChunkT) THEN BEGIN
      WriteLn(M,'==========================================================');
      WriteLn(M,'MIDI track chunk');
      Write(M,'Chunk type  : ',(MIDChunkInfo(MIDBuffer^).ckID));
      WriteLn(M,'(',ActSize,' Bytes)');
      WriteLn(M,'----------------------------------------------------------');
      MIDIncrementPtr(MIDBuffer,SizeOf(MIDChunkInfo));
      SaveStart := MIDBuffer;

{ New chunk has been started, so set MIDLoclSize to 0 }
      MIDLoclSize := 0;

      WHILE (MIDLoclSize < ActSize) DO BEGIN

{ Read DeltaTime of event }
         MIDReadDeltaValue(MIDBuffer, DeltaVal);
         ActTime  := MIDCalcDeltaValue(DeltaVal);

{ What kind of event is it ? }
         ActData  := BYTE(MIDBuffer^);

{ It is a MIDI event }
         IF (ActData IN [$80..$EF]) THEN BEGIN
            Write(M,'Delta = ',ActTime:7,'-> ');
            MIDScanMIDIEvent(MIDBuffer);
            END;
{ It is a Meta event }
         IF (ActData = $FF) THEN MIDScanMetaEvent(MIDBuffer);
{ It is a System Exclusive event }
         IF (ActData IN [$F0, $F7]) THEN MIDScanSysExEvent(MIDBuffer);
         END;
      END
{ No chunk recognized }
   ELSE
    MIDSkipChunk(MIDBuffer);
   END;

FUNCTION MIDBuildScript(MIDFilename : String; Flag : BYTE):BOOLEAN;
{
 * INPUT    : Filename of desired MIDI file as string
 * OUTPUT   : None
 * PURPOSE  : Creates script file in SMF format from MIDI file, in
 *            which the events are broken down into details.
}
VAR
   MIDBuffer : Pointer; { Variable for MID data in memory    }
   MIDIntern : Pointer; { Copy of variable above for processing }
   Check     : BOOLEAN;
   WorkStr   : String;
```

```
BEGIN
   IF (Flag = 1) THEN BEGIN
      IF (Pos('.',MIDFileName) > 0) THEN
         WorkStr := Copy(MIDFileName,1,Pos('.',MIDFileName)-1);
      Assign(M,WorkStr+'.TXT');
      END
   ELSE AssignCrt(M);

   ReWrite(M);

   MIDBuildScript := FALSE;
{ Read MID file into memory }
   Check := MIDGetBuffer(MIDBuffer, MIDFilename);

{ If error occurred and file was already in memory, release memory }
   IF (Check = FALSE) THEN BEGIN
      IF (MIDErrStat = 220) THEN Check := MIDFreeBuffer(MIDBuffer);
      Exit;
      END;

{ Display header }
   WriteLn(M,'MIDSCRIPT  -  A MIDI File Scripter');
   WriteLn(M,'---------------------------------');
   WriteLn(M,'Name of MIDI file  : ', MIDFilename);
   WriteLn(M,'File size in bytes : ', MIDFileSize);
   WriteLn(M);

{ Initialize global variable }
   MIDGlobSize := 0;
{ Make copy of pointer variable }
   MIDIntern := MIDBuffer;

{ Until end of data has been reached, continue taking data }
{ from memory and interpret it according to SMF standards. }
   REPEAT
      MIDInterpretChunk(MIDIntern);
      WriteLn('   ',(MIDGlobSize*100/MIDFileSize):3:2,'%  finished');
      Gotoxy(1, WhereY-1);
      UNTIL (MIDGlobSize >= MIDFileSize);
   WriteLn;
{ Finished working, free memory }
   Check := MIDFreeBuffer(MIDBuffer);
   Close(M);
   IF (Check = TRUE) THEN MIDBuildScript := TRUE;
   END;

BEGIN
   MIDErrStat := 0;
   END
```

MIDSCRIP

The MIDSCRIP.PAS program is a tool that allows you to access the MIDBuildScript function from the MIDTOOL unit.

The program checks whether you've specified a filename with the program call at the DOS command line, and whether you've specified screen output or output to a text file.

Watch your memory limit

It's important to set the upper memory limit of your own programs, since memory won't be available outside of the program limits. However, this memory is needed for loading the data contained in an SMF file. In this example, we've reserved 16K for the stack and 64K for the program.

If you didn't include any parameters with your function call, a short help text explaining the program's syntax is displayed.

To keep the program simple, it works only with files that have the .MID filename extension. All files with other extensions are simply ignored. SMF files should, for clarity and organization, always have the .MID filename extension.

After the /S parameter, which indicates screen output, has been checked, MIDBuildScript is started along with the specified filename and the flag for either screen or text file output.

If the function value isn't TRUE, then PrintMIDErrStatus is used to display the appropriate error message, and the program is ended with the corresponding error level. Otherwise, a text is displayed indicating that a text file has been created.

With this program, you can evaluate the contents of any SMF file.

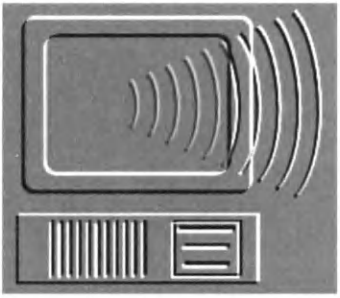
Since the evaluation produces a lot of text, a 150K MIDI file could result in a 2 Meg script file. So you don't think that the program has crashed during the evaluation of such a long file, MIDBuildScript constantly indicates, on the screen, which percentage of the file has already been evaluated.

Sound Blaster Book Companion diskette

However, MIDI files that are this large are relatively rare. Usually they are much shorter, so the script files will also be less lengthy.

Try using this program with your favorite song.

```
Program MIDScript;
{* Demo program for MIDTOOL unit: Creating a script file *}
{ Limit memory for DOS allocation }
{$M 16384,0,65536}
Uses Crt, MIDTool;
VAR
   Check   : BOOLEAN;
   Flag    : BYTE;
   WorkStr : String;
BEGIN
  ClrScr;
  WriteLn('MIDScript v1.0 (C) 1992 Abacus - Author: Axel Stolz');
{ No command line parameter given? display help text }
   IF (ParamCount = 0) THEN BEGIN
      WriteLn('Syntax      : MIDSCRIP midfile[.MID] [/S]');
      WriteLn('              midfile[.MID] = Standard MIDI file');
      WriteLn('                        [/S] = Output to screen');
      HALT(0);
      END;
{ Accept command line parameters }
   WorkStr := ParamStr(1);
   IF ((ParamStr(2) = '/S') OR (ParamStr(2) = '/s')) THEN
      Flag := 0
   ELSE
      Flag := 1;
{ If extension not entered, add ".MID" extension }
   IF Pos('.',WorkStr) > 0 THEN
      WorkStr := Copy(WorkStr,1,Pos('.',WorkStr)-1);
  WriteLn;
  Check := MIDBuildScript(WorkStr+'.MID',Flag);
  WriteLn;
  IF NOT(Check = TRUE) THEN BEGIN
     Write('Error #',MIDErrStat,' ');
     PrintMIDErrMessage;
     Halt(MIDErrStat);
     END;
  IF (Flag = 1) THEN
     WriteLn('Result stored in ',WorkStr,'.TXT.');
  END.
```



Chapter 6

Sound Card Compatibility

Not all sound cards are the same. Developers try to add unique features to their cards to distinguish them from other sound cards. Unfortunately, this creates problems for users. Programs that run with one type of sound card may not run with others. Even cards that claim to be compatible with other cards may be only partially compatible.

In this chapter we'll provide a brief description of some sound cards. If you need additional information, contact the appropriate manufacturer.

6.1 Sound Blaster MultiMedia Upgrade

Creative Labs, Inc.
1901 McCarthy Boulevard
Milpitas, CA 95035
Tel: (408) 428-6600
FAX: (408) 428-6611
Suggested List Price: $799.95

This is probably the most complete package available. It turns your 386 (or higher) computer into a complete Multimedia System. The software included on the companion diskette for this book is compatible with the sound card.

System Requirements:

- IBM compatible 386SX or higher
- Minimum 2MB RAM
- DOS 3.1 or higher

- VGA Graphics Adapter
- 30MB hard drive
- 3.5 inch, 1.44MB floppy drive
- Mouse

Features:

- Stereo recording; 4K to 44.1KHz sampling rates
- Mono compatible with AdLib and Sound Blaster
- Enhanced 4-Operator OPL3 Stereo FM Synthesizer
- Stereo Digital/Analog Mixer - stereo FM DAC, line-in, microphone, CD Audio and PC's internal speaker
- Software control over fade, pan, left and right volume microphone mixing
- MIDI adapter included
- High speed, high performance CD-ROM Interface (for use with Creative, Panasonic and Matsushita Drives only) with internal connector for CD-Audio mixing
- Joystick port
- Microphone amplifier with AGC (Automatic Gain Control)
- Four watt per channel output amplifier (PMPO)
- Jumper selections for IRQ, DMA and I/O address
- Connector pins for PC's internal speaker, line out, microphone, CD-ROM interface and CD-Audio in
- Full compliance with Microsoft Multimedia Level I Extensions to Windows

Bundled with:

- MS Windows 3.1
- MS Bookshelf (CD-ROM Reference Library)
- MS Works for Windows

- Special Edition Tempra from Mathematica
- Sherlock Holmes, Consulting Detective (game)
- MacroMind Action!
- Authorware Star
- Creative Sounds
- Creative Music Clips
- MIDI Adapter and Sequencer
- FM Intelligent Organ
- SB Voice Editor and Utilities
- SBTALKER with Dr. SBAITSO
- CD Music Player
- MMPlay Presentation
- DOS Drivers and Windows DLL
- SB MIDI (MIDI file driver)
- SBSIM (Sound Blaster Standard Programming Tools)
- MMPlay Utility

Package includes:

- Creative CD-ROM
- Sound Blaster Pro Card
- RCA Cable
- MIDI Cables
- 5.25 and 3.5 inch diskettes
- Manuals

6.2 Sound Blaster Pro

Creative Labs, Inc.
1901 McCarthy Boulevard
Milpitas, CA 95035
Tel: (408) 428-6600
FAX: (408) 428-6611
Suggested List Price: $299.95

The software included on the companion diskette for this book is compatible with this sound card.

System Requirements:

- IBM PC/AT or 100% 286 and higher compatibles (286 and higher recommended)
- Minimum 512KB RAM
- DOS 3.0 or higher
- EGA or VGA (VGA Recommended)

Features:

- Stereo recording; 4K to 44.1KHz sampling rates
- Mono compatible with AdLib and Sound Blaster
- Enhanced 4-Operator OPL3 Stereo FM Synthesizer
- Stereo Digital/Analog Mixer - stereo FM DAC, line-in, microphone, CD Audio and PC's internal speaker
- Software control over fade, pan, left and right volume microphone mixing
- MIDI adapter included
- High speed, high performance CD-ROM Interface (for use with Creative, Panasonic and Matsushita Drives only) with internal connector for CD-Audio mixing
- Joystick port

- Microphone amplifier with AGC (Automatic Gain Control)
- Four watt per channel output amplifier (PMPO)
- Jumper selections for IRQ, DMA and I/O address
- Connector pins for PC's internal speaker, line out, microphone, CD-ROM interface and CD-Audio in
- Full compliance with Microsoft Multimedia Level I Extensions to Windows

Bundled with:

- MIDI Adapter and Sequencer
- FM Intelligent Organ
- VEDIT Voice Editor and Utilities
- SBTALKER with Dr. SBAITSO
- CD Music Player
- MMPlay Presentation
- DOS Drivers and Windows DLL
- SB MIDI (MIDI file driver)
- SBSIM (Sound Blaster Standard Programming Tools)

Package includes:

- Sound Blaster Pro Card
- RCA Cable
- MIDI Cables
- 5.25 and 3.5 inch diskettes
- Manual

6.3 Sound Commander fx

MediaSonic, Inc.
46726 Fremont Boulevard
Fremont, CA 94538
Phone: (510) 438-9996
FAX: (510) 438-9979
Suggested List Price: $139

The software included on the companion diskette for this book is compatible with this sound card only if you are able to obtain the Creative Lab drivers for the Sound Blaster card.

System Requirements:

- PC XT AT, 286 or higher
- 512K RAM for DOS software
- DOS 3.0 or higher
- 2MB RAM for Windows software

Features:

- Compatible to Adlib, Covox Speech Thing, Sound Blaster
- Karaoke Mixing
- Full MIDI interface
- 11 Voice Yamaha SFM Sound Chip
- Socket for adding Yamaha FM synthesizer chip to upgrade to full stereo 22 voices
- Digital to Analog Converter
- Compatible to Sound Blaster (in mono mode)
- Infra-Red Remote control
- Output power amplifier capable of driving small speakers (4 Watts per channel)

Bundled with:

- CD Studio
- Media Karaoke
- Media SoundTrack
- ShowPartner Lite

6.4 Sound Commander Gold

MediaSonic, Inc.
46726 Fremont Boulevard
Fremont, CA 94538
Phone: (510) 438-9996
FAX: (510) 438-9979
Suggested List Price: $239

The software included on the companion diskette for this book is compatible with this sound card. However, for some programs you may need to obtain Sound Blaster drivers from Creative Labs.

System Requirements:

- PC, XT AT, 286 or higher
- 512K RAM for DOS software
- DOS 3.0 or higher
- 2MB RAM for Windows software

Features:

- Fully Multimedia PC compatible
- AdLib compatible
- Covox Speech Thing compatible
- Sound Blaster compatible

- Stereo Synthesizer:

  Yamaha FM Synthesizer chip

  20 independent channels

  4 Operators, 8 Waveforms FM Synthesis

  18 Timbres (9 per channel)

- Analog Mixer Multisource: Internal FM Synthesizer, Internal CD Audio, Digitally sampled Audio, External line-in, Microphone input, Internal PC Speakers, Programmable 24-level volume control for each input

- Stereo Digital to Analog Converter

  Sound Blaster compatible (in mono mode)

  Digitized Music, Voice and Sound Effects are played back in full stereo

  Selectable sampling rates (4K to 44.1K)

  DMA Selectable by jumper

- Stereo Analog to Digital Converter

  Sound Blaster compatible (in mono mode)

  Stereo recording from microphone, stereo line-in and CD-Audio

  Selectable sampling rate (4K to 44.1K)

  DMA Selectable by jumper

- DSP RAM Buffer

  8K DSP RAM for PCM digitized sound and MIDI output

  Programmable for updating digitized sound and MIDI functions

  Programmable storage of digitally sampled instruments

- MIDI Interface

  Standard UART MIDI interface with *K DSP RAM buffer built-in

  Connect MIDI instruments directly

- Output Power Amplifier

  Hardware Master Output Volume Control Knob

  Software Master Volume Control

  4 Watt per channel

- Infra Red remote control

  Permits remote control

  Software permits control of home appliances

Bundled with:

- CD Studio
- Karaoke Software
- SoundTrack Utilities
- ShowPartner Lite
- Sound Commander Mixer software

6.5 Sound Commander MultiMedia

MediaSonic, Inc.
46726 Fremont Boulevard
Fremont, CA 94538
Phone: (510) 438-9996
FAX: (510) 438-9979
Suggested List Price: $699

The software included on the companion diskette for this book is compatible with this sound card. However, for some programs you may need to obtain Sound Blaster drivers from Creative Labs.

System Requirements:

- PC, XT AT, 286 or higher
- 512K RAM for DOS Software
- DOS 3.0 or higher
- 2MB RAM for Windows software

Features:

- Fully Multimedia PC compatible
- AdLib compatible
- Covox Speech Thing compatible
- Sound Blaster compatible
- Stereo Synthesizer:

  Yamaha FM Synthesizer chip

  11 independent channels

  18 Timbres (9 per channel) when upgraded

- Analog Mixer Multisource: Internal FM Synthesizer, Internal CD Audio, Digitally sampled Audio, External line-in, Microphone input, Internal PC Speakers, Programmable 24-level volume control for each input
- Stereo Digital to Analog Converter

  Sound Blaster compatible (in mono mode)

  Digitized Music, Voice and Sound Effects are played back in full stereo

  Selectable sampling rates (4K to 44.1K)

  DMA Selectable by jumper

- Stereo Analog to Digital Converter

    Sound Blaster compatible (in mono mode)

    Stereo recording from microphone, stereo line-in and CD-Audio

    Selectable sampling rate (4K to 44.1K)

    DMA Selectable by jumper

- DSP RAM Buffer

    8K DSP RAM for PCM digitized sound and MIDI output

    Programmable for updating digitized sound and MIDI functions

    Programmable storage of digitally sampled instruments

- MIDI Interface

    Standard UART MIDI interface with *K DSP RAM buffer built-in

    Connect MIDI instruments directly

- Output Power Amplifier

    Hardware Master Output Volume Control Knob

    Software Master Volume Control

    4 Watt per channel

- Infra Red remote control

    Permits remote control

    Software permits control of home appliances

Bundled with:

- CD Studio
- Karaoke software
- SoundTrack utilities
- ShowPartner Lite

6.6 Sound Master II

Covox, Inc.
675 E. Conger St.
Eugene, OR 97402
Phone: (503) 342-1271

Sound Master II is not compatible with any of the programs in this book. This sound card can be used with games that are AdLib compatible. We've included Sound Master II in this list because of its other interesting features.

System Requirements:

- IBM AT/286/386/486 or compatible
- 256K RAM
- DOS 3.1 or higher
- Compatible with AdLib, Speech Thing, Voice Master, most games

Features:

- Real time data compression
- 4 Watt amplifier
- 11 voice music synthesizer
- DMA digitized recording and playback of up to 25,000 samples per second
- Special effects program
- Built-in MIDI port
- Low noise, high sound quality
- Improved internal PC speaker sound

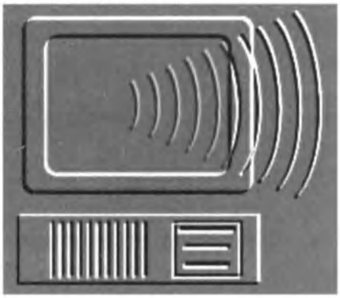
Package includes:

- Sound Master II card

- Windows multimedia extensions
- MIDI cables
- User and reference manuals
- Installation utilities
- Voice Command Software
- Developer Libraries
- Microphone/Headset

Bundled with:

- Voice Master Key (voice recognition software)
- SMulator Sound card emulation utility
- SMII - Configure and test sound system
- PC-Lyra - Music program
- VMEdit - Waveform editing utility



Appendix A

Glossary

The following is a list of music terms that are often used with Sound Blaster. When you use Sound Blaster to create your own music, you'll encounter terms from different fields, such as music, computers, and synthesizers.

This glossary contains music terms that may be unfamiliar to computer users and computer terms that may be unfamiliar to musicians. If you need more information about a term, refer to the index following the glossary.

A440
Also called *Concert A*. This pitch, which sounds at 440 Hz, is the usual tuning note used by orchestras. See also **Concert A**.

Accidental
A symbol used to raise or lower a note above or below its written pitch.

Acoustic instrument
An instrument that vibrates to produce sound directly in the air without electronic means.

ADC
Acronym for Analog to Digital Converter. The ADC scans or samples the analog signal (e.g., a sample of a live sound) at predetermined intervals and converts the results of these samples into numeric values (a series of zeros and ones).

AdLib card
An early sound card. The introduction of the AdLib card coincided with the rapidly growing market for PC-based computer games.

Software firms that had provided music for the Roland and IBM cards also began working with AdLib. The available software support and its affordable price made the AdLib card the standard for sound cards intended for non-professional users.

ADSR envelope generator A generator with four separate stages (attack, decay, sustain, release) in the envelop it generates. See also **Attack, Decay, Sustain, Release**.

Aftertouch Refers to the degree of pressure exerted on a key during play, until it is released.

The two types of aftertouch include monophonic and polyphonic. A keyboard with monophonic aftertouch senses only the hardest pressed key, while an instrument with polyphonic aftertouch can sense the pressure on each key individually when a chord is played.

This latter feature is found only on more expensive keyboards. MIDI measures both forms of aftertouch in 128 increments.

Algorithm A specific combination, in FM sound synthesis, of a synthesizer's operators used to create a patch.

Amplitude Amplitude is determined by the highest point along the curve of the sound wave. The higher the amplitude, the louder the sound. The physical unit of loudness is the decibel (dB). Decibels are a logarithmic unit of measure, specifying the degree of loudness of the wave. See also **Decibel**.

Analog to Digital Converter See **ADC**.

Attack The attack or rise of a tone determines the amount of time required for the tone to reach its maximum amplitude. See **Amplitude**.

Audio signal An electric signal of different voltage that becomes sound when amplified and fed to a speaker.

Bass guitar synthesizer A specialized guitar synthesizer designed to work with a bass guitar controller.

BBS Abbreviation for Bulletin Board System, which is used for exchanging files, electronic mail (EMAIL) and messages. Some BBS system operators (SYSOPS) have dedicated their systems to Sound Blaster public domain software and shareware.

Bit The smallest unit in the binary number system. It can assume only two states (0,1) and, therefore, store only two different pieces of information. To store a character, several bits must be combined into a byte.

Byte A group of eight bits. While a bit can assume only two states, 0 and 1, a byte can store values from 0 to 255.

CD Abbreviation for Compact Disc. The CD is now used in audio recording (CDs are smaller, less fragile, and more profitable than vinyl records), and in computing.

Much of the software items and sound files included with the Sound Blaster MULTIMEDIA UPGRADE KIT™ are on CD and can be used with the CD player included with the KIT.

Channel message A MIDI message that can only be received by MIDI devices in a MIDI network. These MIDI devices must be set to the same MIDI channel as the sending device.

Channel mode message A channel message that carries data about the MIDI receiving mode.

Channel voice message A channel message that carries performance data between devices.

Chord A set of pitches played simultaneously. Chords are displayed vertically, with one note above the other.

Chorusing An audio effect that changes the audio signal so one instrument will sound like several.

Chunks A RIFF file consists of several smaller file components called chunks. A chunk is a logical data segment that always has the same structure, regardless of what type of data it contains.

Clef Refers to a symbol that usually appears at the beginning of a staff. It shows the pitch range where the notes on the staff fall. A bass clef shows a low pitch range. A treble clef shows a high pitch range.

Concert A In the music world, the frequency of 440 Hz is very important. This frequency defines the concert pitch a1 and is the frequency to which most concert instruments are tuned.

Creative Music File format Standard Sound Blaster file format for FM synthesizer playback. A Creative Music File can usually be recognized by the .CMF extension.

The special characteristic of CMF is that its music block is identical to the MIDI format specified by the IMA. Therefore, the music information stored in CMF files follows MIDI specifications.

Crescendo A gradual increase in volume.

CT-Voice file format Standard Sound Blaster format for file playback. A CT-Voice file can usually be identified by the .VOC extension. A VOC file is divided into two blocks: the header and the actual data.

CT-VOICE.DRV The CT-VOICE.DRV driver is part of the Sound Blaster software package. This driver is a small but extremely powerful software tool that allows you to access all the important functions of your sound channel (such as for playing VOC files).

DAC Acroynym for Digital to Analog Converter. A Digital to Analog Converter reverses the conversion that was previously made by the ADC.

The DAC converts the digital sound (a series of zeros and ones) back into an analog signal. However, instead of being smooth and continuous like the original analog sound, this reproduced analog signal consists of individual steps.

Decay The decay determines the amount of time required for the tone to fall from its maximum loudness to an amplitude level that is then sustained for a certain period.

Decibel A unit of measurement used to measure amplitude.

Decrescendo A gradual decrease in volume.

Delay An audio effect that delays a version of the incoming audio signal to create a synthetic echo or reverberation.

Digital Information stored or sent as a numerical value.

Digital to Analog Converter
A device used to convert digital numbers to continuous analog signals. See also **DAC**.

DIN plug
A standard type of plug at each end of a MIDI cable.

DMA-Channel
An acronym for "Direct Memory Access." This refers to Sound Blaster's ability to access your computer's memory directly without going through the CPU. This saves computing time and also allows sound processing to occur along with other tasks.

Drum machines
These devices act as rhythm generators producing sounds of various drum types and other percussion sounds. Like melodic instrument voices, these sounds can also be sent over MIDI. Also called *drum computers*.

Duration
Length of a musical note or combination of notes.

Dynamic marking
An expression printed in a score showing the volume of the music.

Dynamics
Volume or amplitude of a musical note. Dynamic markings commonly used in musical notation, from softest to loudest (with abbreviations in parentheses), are pianississimo (***ppp***), pianissimo (***pp***), piano (***p***), mezzo-piano (***mp***), mezzo-forte (***mf***), forte (***f***), fortissimo (***ff***), and fortississimo (***fff***).

Effect
An audio signal processor that modifies the quality of an audio signal from a synthesizer or other source.

EG
See **Envelope generator**.

Electronic instrument
A musical instrument that uses audio signals, amplification, and speakers to create sound.

Envelope
A graph of frequency, amplitude, or timbre of sound over a specific period of time.

Envelope generator
A module in an analog synthesizer that creates a changing control voltage. This voltage determines the frequency, amplitude, or timbre of each note played by the synthesizer.

Event
Occurrence (e.g., an error) in Microsoft Windows to which a sound can be assigned.

Expanders

A synthesizer that does not include a keyboard. Expanders must be controlled by a master device. Since it contains fewer moving parts, an expander often costs less than its keyboard equivalent.

Also, in large MIDI systems it may be more practical to use expanders because they occupy less physical space than a keyboard synthesizer.

Fade in

Steadily increasing the volume level of a sound from silence to an audible level. Sound Blaster has utilities to add fade in to an existing sound.

Fade out

Steadily decreasing the volume level of a sound until the sound is no longer audible. Sound Blaster has utilities to add fade out to an existing sound.

FM

Abbreviation for Frequency Modulation. Used in broadcasting and, more recently, in music synthesizers. Sound Blaster cards have FM synthesis capability.

FM synthesis

Digital synthesis that combines the outputs of different digital oscillators by using different combinations, called algorithms, to create patches. See also **Algorithms**.

Forte (f)

Refers to loud music.

Fortissimo (ff)

Refers to very loud music.

Fourier analysis

The mathematical process dividing a sound into a finite number of different sine waves.

Fourier synthesis

The reverse of Fourier analysis. Fourier synthesis can be used to produce specific sounds artificially.

For example, it's possible to produce an instrument sound by taking a sine wave as a base tone and adding the required number of overtones. In this way you can digitally produce a natural sound without having previously recorded the sound (i.e., sound sampling).

Frequency
Refers to the rate of vibration. How high or low a given tone sounds depends on the number of pulses per second. This number of pulses is referred to as the tone's *frequency*. The unit of measure for frequency is Hertz (Hz). This unit specifies the number of pulses per second that a given tone emits.

Fundamental frequency
The lowest possible frequency.

Hertz (Hz)
Frequency in cycles per second of a musical note or radio wave. One cycle equals one Hertz. Average human hearing range is from 20 Hz to 20000 Hz.

IBM Music Feature card
One of the original sound cards. Although fairly expensive, this card includes an eight-voice stereo synthesizer and a complete MIDI interface. The heart of this card is the Yamaha YM-2164 sound chip. Sound generation occurs through an FM synthesizer with multiple control parameters. There are also 240 preprogrammed sounds that include reproductions of traditional musical instruments. You can have up to four of these cards installed simultaneously, allowing a maximum of 32 simultaneous voices.

IFF Format
Developed by Electronic Arts and used by DPaint. IFF is an acronym for "Interchange File Format". The Commodore Amiga uses IFF graphic files, IFF text files, and IFF sound sample files. However, PCs normally use only the DPaint IFF graphic files. These files are identified by the ".LBM" filename extension.

IMA
Acronym for International MIDI Association. A non-profit U.S. organization that releases information about MIDI specifications.

International MIDI Assoc.
See **IMA**.

Interval
The difference in pitch between two notes.

Keyboard controllers
High-quality keyboards that can send MIDI messages but seldom contain their own system of tone generation. The main responsibility of keyboard controllers is to control a MIDI system.

Keyboard synthesizers	Includes both a keyboard and a tone generator. The general term "keyboard" usually implies a keyboard synthesizer. A good keyboard synthesizer can also be used as the master keyboard.
Line-In	Sound Blaster input jack for connection to a high impedance external audio device (e.g., the LINE OUT jack on a tape deck).
Master device	One of the most important components in a MIDI system. The master device controls all other connected devices. The most common master devices are keyboards or computers.
Mezzo forte (mf)	Refers to moderately loud music.
Mezzo piano (mp)	Refers to moderately low music.
MIDI	Acronym for Musical Instrument Digital Interface (pronounced "middy"). A standard communications protocol for exchanging information between computers and musical synthesizers. MIDI allows interaction between musical instruments and computers. MIDI can instruct instruments to play music, change sound parameters, and more.
MIDI cable	A cable with 5-pin plugs on each end. This cable is used to carry messages between MIDI devices.
MIDI Mapper	Microsoft Windows utility for customizing MIDI setups. MIDI configurations change from user to user, so MIDI Mapper allows fast setup of MIDI channels, sound patches, and keymap configuration.
MIDI channel	Transmission that allows messages to be sent over a MIDI network to single devices without being received by all the devices on the network.
MIDI device	Device featuring MIDI ports and microprocessor cable for sending or receiving MIDI messages.
Mixer	Device used for controlling sound input to a single amplifier, tape recorder, etc. Sound Blaster Pro has software mixers for controlling volume on Microphone, CD player, Line-In, ADC, and FM.

MMSYSTEM.DLL This is a Dynamic Link Library (DLL) and contains an entire row of functions that let you play back sound samples under Windows.

Monophonic aftertouch See **Aftertouch**.

Multimedia Buzzword referring to the combination of music and images controlled by, and output using, a computer. With Multimedia you can create, for example, a computer-generated "slide show" with narration or an actual animated film.

Musical elements See **Pitch, Dynamics, Timbre, Duration**.

Natural A traditional symbol used to cancel the effect of a preceding flat or sharp symbol.

Note off The moment when a key is released.

Note on The moment when a key is pressed.

Octave A standard pitch interval spanning 13 consecutive keys on a music keyboard. The upper pitch in an octave is twice the frequency of the lower pitch.

Operator Refers to the digital oscillator used in FM sound synthesis.

Overtones Sound waves, or frequencies, that occur as multiples of the tone's fundamental frequency. Each instrument emits a specific number of overtones that determine the characteristics of its sound. See also **Fundamental frequency**.

Packing Compression of samples used as a disk space saving measure.

Panning Moving sound from one stereo channel to another (e.g., left to right).

Parallel interface Parallel interfaces exchange data 8 bits at a time.

Patch A specific sound design created using the synthesizers controls. A synthesizer plays notes using the sounds that are defined as patches.

PCM format Acronym for Pulse Code Modulation. It refers to a specific encoding format for sound samples. The PCM format is currently used for WAV files.

Periodic waves Sound waves where the first pulse is followed by an indefinite number of identical pulses. Periodic waves result in sounds that can be called tones, such as the tone of a guitar, a piano, or a bell. An example of a periodic wave is a sine wave. See also **Sine wave**.

Pianissimo (pp) Refers to music of a very low volume. See also **Dynamics**.

Piano (p) Refers to music of low volume. See also **Dynamics**.

Pitch bend A good keyboard should be able to bend pitches the way a horn or guitar does as it slides up or down into a note. This effect is accomplished using a wheel to control the pitch. There is also a MIDI message to control this effect.

Pitch Frequency of a musical note. The higher the frequency, the higher the pitch. For example, the note frequencies 440 Hz and 880 Hz are both A pitches, but the 880 Hz note is one octave higher than the 440 Hz note (see also **A440** and **Octave**).

PORTMENTO The sweeping of a tone up or down the scale, without any noticeable steps.

Public domain Refers to software made available to users free of charge.

Release The release of a tone is a second fade; the tone actually fades away completely until the amplitude of its pulses has reached zero. The tone's release determines the time its amplitude needs to reach zero.

Rest A symbol showing a period of silence.

RIFF format Microsoft has created a basic file structure for most of the file types used with Windows. This makes it easier to use different formats simultaneously.

If you've worked with the DPaint application, you've already used files that conform to this structure. Electronic Arts has developed a system called IFF, that is also used by DPaint. IFF is an abbreviation for "Interchange File Format".

The Commodore Amiga uses IFF graphic files, IFF text files, and IFF sound sample files. However, PCs usually use only the DPaint IFF graphic files, which are identified by the ".LBM" filename extension. Microsoft used the concept of IFF when developing RIFF (Resource Interchange File Format). A RIFF file consists of several smaller file components called chunks. See also **Chunks**.

Roland LAPC-1

A professional sound card featuring eight-voice polyphonic and 128 preprogrammed sounds, that can also be modified. The LAPC-1 also provides a drum computer and digital effects device.

These features, however, make this card too powerful to be used for only games. By using an add-on MIDI box, you can produce professional sound applications. However, remember that, in price and quality, the available software and hardware extensions dramatically increase this card's power.

Sampling frequency

The interval at that the ADC scans or samples the analog signal at predetermined intervals and converts the results of these samples into numeric values.

Sampling rate

See **Sampling frequency**.

Sampling

This is the process of digitizing sounds. This method allows a sound consisting of an analog signal to be transformed into digital data (i.e., bits and bytes).

This task is performed by an Analog to Digital Converter (ADC). Sound Blaster lets you sample sound from prerecorded sources (e.g., CD player) or directly through a microphone. You can then store this sound sample to disk for later recall. See also **ADC**.

SBFMDRV.COM

A driver for the Sound Blaster card's FM voices that is included in the Sound Blaster package.

SBFMDRV.COM

Driver file for FM sound playback.

SBI format

An acronym for "Sound Blaster Instrument".

Scales

The modern musical scale consists of seven base tones and five half tones. The seven base tones are located either a whole step from the adjacent note (C-D, D-E, F-G, G-A, A-B) or a half step from the adjacent note (E-F, B-C). Each note can be raised or lowered by a half step.

Score

A musical composition that is stored in a sequencer or in a computer (see also **Sequencer**). A traditional score is a printed version of a musical composition using traditional musical notation.

Sequencer

MIDI device used for recording, saving, and playing back musical compositions. Many sequencers allow you to record individual channels while playing back existing channels.

The Sound Blaster MIDI Kit includes a MIDI interface and sequencer software that accesses both MIDI instruments and the Sound Blaster FM synthesizer hardware.

Shannon theorem

A formula created by the mathematician Shannon. The sample frequency must be exactly double the highest sound frequency that will be digitized. If this requirement is fulfilled, all necessary information on the analog signal will be stored digitally.

For example, if you want to sample a frequency of 14000 Hz, you must use a sampling rate of 28000 Hz to achieve the best results. See also **Sample frequency**.

Shareware

Software made available to the public on a "try-before-you-buy" basis. If the user finds the software useful, he or she can send the author a fee to register the software.

Sine wave

A wave whose first pulse is followed by an indefinite number of identical pulses. These waves result in sounds that can be called tones, such as the tone of a guitar, a piano, or a bell. A sine wave is a periodic wave. See **Periodic waves**.

Slave devices

Refers to all the MIDI system components , such as expanders, effects devices, and drum machines that are controlled by the master device.

Sound waves
Waves that are created when air is pulsated. These waves are then registered as sound. The amplitude and frequency of these waves determine what type of tone is heard.

Soundtracker format
Probably the most popular music file format used on Amiga systems today. This format uses digitized instrument samples. It modifies their playback frequency to bring the samples to the desired pitch.

This format lets you create very realistic musical pieces since sampled instruments sound much more realistic than synthesized ones. Also, since each instrument must be loaded into memory only once, this results in much shorter files than fully digitized pieces of music.

Sustain
An effect that allows notes to continue to sound even after you release the keys that you pressed to start them.

Sustain Level
The amplitude at which the tone is emitted for a certain amount of time before it finally fades away.

Tempered scale
In this scale, the intervals between the half steps are always equal. Therefore, D sharp and E flat actually have the same frequency and can be played on the same key.

Tempo
Speed of a musical composition, listed in beats per minute. A tempo of 60 refers to 60 beats per minute, or one beat per second. Most sheet music lists tempo markings as words (e.g., *Allegro*).

Timbre
(pronounced "TAM burr") Timbre refers to the tone color of a note. A note played on a flute has a different timbre from the same note played on a violin because of instrument construction, method of tone production, etc.

Velocity
Refers to the manner and speed with which the key, drum pad, or string producing a sound is activated. In other words, the velocity (speed) at which the player attacks a note defines the volume of the note.

Most MIDI keyboards are velocity-sensitive, which means that the faster (harder) the user plays a note, the louder that note will sound. The MIDI system provides 128 levels for velocity.

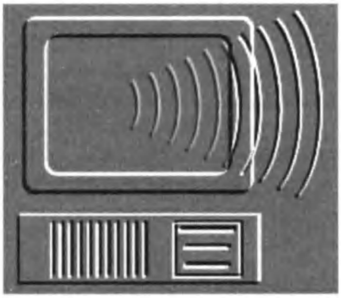
This is slightly limited when compared to the much finer nuances that are possible with acoustical instruments.

VOC files See **CT-Voice file format**.

WAV files Sound files as used in Microsoft Windows. A WAV file can be identified by the .WAV extension.

WAVE format A format developed by Microsoft for describing RIFF file structures.

Waveform The shape of a sound wave, varying in complexity with the type of sound.



The Sound Blaster Book

Index

A-C

M

N-P

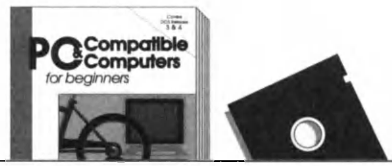
R

S

T

U-V

W



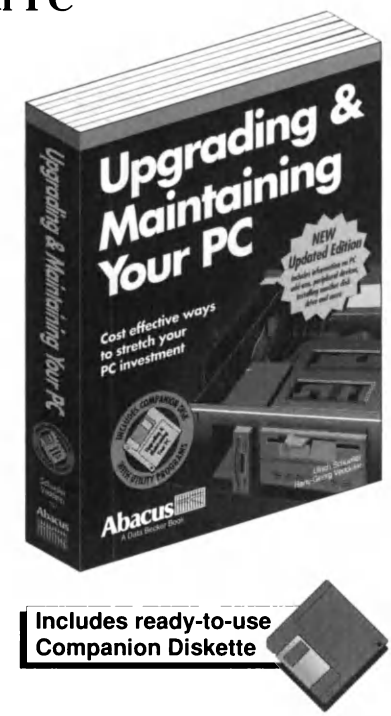
Abacus pc catalog

Order Toll Free 1-800-451-4319

To order direct call Toll Free 1-800-451-4319

Productivity Series books are for users who want to become more productive with their PC.

Upgrading & Maintaining Your PC



Upgrading & Maintaining Your PC provides an overview of the fundamental and most important components of a personal computer. It describes PC and compatible computers from the outside to the inside. **Upgrading & Maintaining Your PC** also introduces you to the relationship between individual components to help you get to know your computer better.

PCs are a major investment. **Upgrading & Maintaining Your PC** will show you how to turn your PC into a high perfomance computing machine. This book describes what you will see when you open the "hood" and how all the parts work together. You'll learn how to add a hard drive, increase your computer's memory, upgrade to a higher resolution monitor or turn an XT into a fast AT or 386 screamer. **Upgrading & Maintaining Your PC** shows you how to do so, easy and economically, without having to be an electronics wizard.

System Sleuth Analyzer from Dariana Technology Group, Inc. is on the companion disk bundled with this guide. *System Sleuth Analyzer* is a $99.95 toolbox of valuable PC diagnostic aids rolled into a single, easy to use software utility. System Sleuth lets you explore exacting details of your PC without the fear of accidental, unrecoverable modifications to a particular subsystem. The menu-driven interface displays all aspects of machine information for easy recognition.

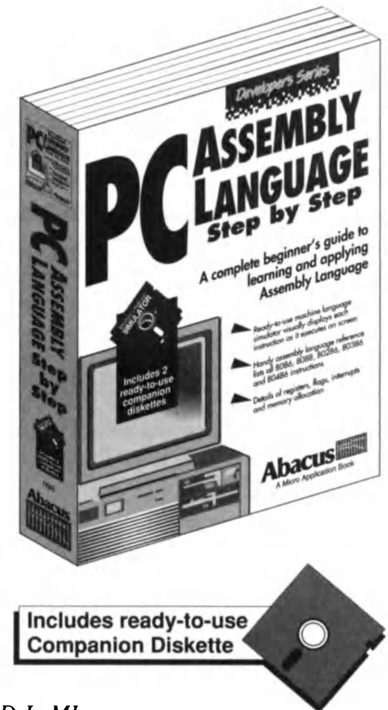
Authors: Ulrich Schuller and Georg Veddeler
Order Item #B167. ISBN 1-55755-167-7.
Suggested retail price $34.95 with companion disk. Canadian $44.95.

To order direct call Toll Free 1-800-451-4319

In US and Canada add $5.00 shipping and handling. Foreign orders add $13.00 per item.
Michigan residents add appropriate sales tax.

Developers Series books are for professional software developers who require in-depth technical information and programming techniques.

Assembly Language Step by Step



For lightning execution speed, no computer language beats assembly language. This book teaches you PC assembly and machine language the right way - one step at a time. The companion diskette contains a unique simulator which shows you how each instruction functions as the PC executes it. Includes companion diskette containing assembly language simulator.

"I've yet to find a book which explains such a difficult language so well. This manual makes Assembly easy to learn." K.B., SC

"The book is a very practical introduction to PC assembly language programming. Anybody without computer know how can understand it." D.J., MI

"Very impressed with this book -Better than all the others I bought." C.S., CA

"This book is an exceptional training tool." J.L., TN

"Good, easy to understand book." S.S., NM

"An excellent introduction to assembly language." A.V., Austrailia

"Excellent, just the book I was looking for, easy to learn from." J.S., PA

"I learned a lot from this well written book." D.Z., OH

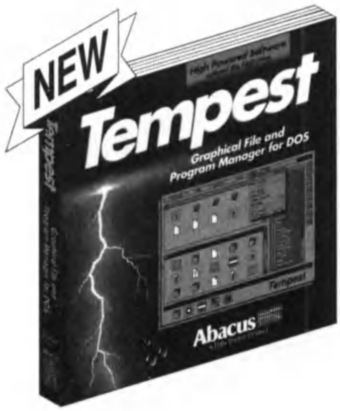
"Outstanding book! Incredibly clear, concise format. To the point!" J.C., HI

Order Item #B096. ISBN 1-55755-096-4.
Suggested retail price $34.95 with companion diskette.
Canadian $44.95.

To order direct call Toll Free 1-800-451-4319

In US and Canada add $5.00 shipping and handling. Foreign orders add $13.00 per item.
Michigan residents add appropriate sales tax.

High Powered Software



Tempest

Graphical File and Program Manager for DOS.

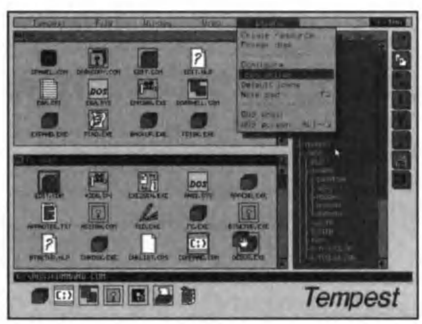
Tempest provides a graphical Program Manager and Shell for all DOS computers. It's the graphic shell alternative to DOS's DOSSHELL. Using pull-down menus, dialog boxes and windows, **Tempest** makes working with DOS easier than ever. All DOS commands are executed with just a click of your mouse button. Whether you need to move files, rename directories or launch programs, you'll find **Tempest** is the answer to easier computing.

Tempest features:

- Intuitive graphical user interface
- Automatically creates Icons for all your files
- Select files from a directory tree by pressing a button
- Automatic allocation of finished icon sets to standard programs
- Copy, move and delete files *and directories* without entering complicated DOS 5.0 commands
- Includes TED - the graphical text editor for DOS 5.0
- Includes 2 screen savers to prevent monitor "burn-in"
- Works with DR DOS 6.0

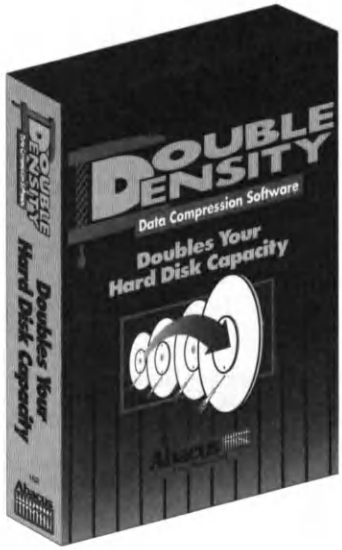
System requirements: 100% compatible IBM XT/AT, 386 or 486 computers. EGA or VGA graphics card required. Hard drive and mouse recommended.

Item #S150. ISBN 1-55755-150-2. Suggested retail price $29.95 with $3^1/_2$" diskette. Canadian $39.95.



To order direct call Toll Free 1-800-451-4319

In US and Canada add $7.50 shipping and handling. Foreign orders add $13.00 per item.
California and Michigan residents add appropriate sales tax.



Double Density

Doubles your hard disk storage space!

DoubleDensity is a hard disk doubler for DOS. **DoubleDensity** increases the storage capacity of any hard drive. For example increasing a 40 meg hard drive to 80 meg, or a 200 meg hard drive to a 400 meg hard drive. At the same time, **DoubleDensity** increases the speed of read accesses on faster computers.

DoubleDensity does not require any additional hardware purchases. Thanks to the fully automatic installation program anyone can install **DoubleDensity** quickly and easily. You don't even need to open the PC.

DoubleDensity:

- Increases the storage capacity of your hard drive up to twice its original capacity
- Increase the speed of read accesses of 386's and higher
- Is easy to install because installation is fully automatic
- Offers continued availability of all DOS commands such as DIR or UNDELETE
- Works with system programs such as SpeedDisk, Compress, DiskDoctor and many cache programs
- Is fully compatible with Windows 3.0 and 3.1
- Takes up no space in main memory with MS-DOS 5.0
- Provides you with the option of protecting your data with passwords
- Uses approximately 47K and can be loaded into high memory

System requirements:
IBM 100% compatible XT, AT, 386 or 486 (recommend minimum AT 12 MHZ).
Microsoft DOS 3.1 or higher.

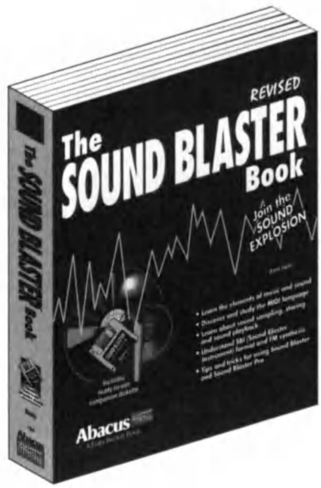
Order Item #S152. ISBN 1-55755-152-0. Suggested retail price $79.95.
Canadian $99.95

To order direct call Toll Free 1-800-451-4319

In US and Canada add $7.50 shipping and handling. Foreign orders add $13.00 per item.
Michigan residents add appropriate sales tax.

The Companion Diskette

The SOUND BLASTER Book



The companion diskette features executable applications created using Borland C++ Version 3.1, Borland C++ for Windows Version Turbo Pascal Version 6.0, Turbo Pascal for Windows and Microsoft Visual Basic.

You'll find applications for playing VOC files and WAV files, for converting Soundtracker MOD files and standard MIDI files to text and many VOC and WAV samples.

The companion diskette saves you time because you don't have to type in the program listings presented in the book. Programs that are on the companion diskette are indicated with a picture of a diskette next to the listing.

Installing the companion diskette

To install the companion diskette, insert the diskette in the drive. Log to that drive and type the following:

```
INSTALL <Enter>
```

You'll find the following directories on the diskette:

ABACUS directory

- Contains the SBBOOK directory, in which the companion diskette files are stored.

SBBOOK directory

- Contains directories according to language or program function.

C directory

- Contains source codes and executable programs written using Borland C++ Version 3.1. CMFDEMO plays a CMF file; VTTEST plays VOC files.

CPPW directory

- Contains source codes and an executable program written using Borland C++ Version 3.1. CPPWSOUND plays WAV files from within Microsoft Windows.

TP60 directory

- Contains source codes and executable programs written using Turbo Pascal 6.0.

ARRANGER plays a series of VOC files; CMFDEMO plays a CMF file; MIDSCRIP converts a standard MIDI file to text;

MODSCRIP converts a Soundtracker file to text; and VTTEST plays VOC files.

TPW directory

- Contains source codes and executable programs written using Turbo Pascal for Windows. TPWSOUND plays WAV files from within Microsoft Windows.

VB directory

- Contains project data and an executable program written using Microsoft Visual Basic. VBSOUND plays WAV files from within Microsoft Windows.

VOCSHELL directory

- Contains the VOCSHELL program, which is used for playing VOC files (included).

WAV directory

- Contains WAV files used by CPPSOUND, TPWSOUND, and VBSOUND.

WAVEPOOL directory

- Contains the WPL.EXE (WAVEPOOL) shareware program and a sample file.

ADVANTAGES...
of book /companion diskette packages:

✔ Save hours of typing in source listings from the book.

✔ Provide complete, ready-to-run executable applications and examples.

✔ Help avoid printing and typing mistakes.

Sound Blaster Companion Diskette

SBBOOK directory

This directory is located in the ABACUS directory. It contains the directories with the applications and program listings found in this book:

- The C directory and the CPPW directory contain source codes and executable programs written with Borland C++ Version 3.1.
- The TP60 directory contains source codes and executable programs written using Turbo Pascal 6.0.
- The TPW directory contains source codes and executable programs written using Turbo Pascal for Windows.
- The VB directory contains an executable program written with Visual Basic.
- The VOCSHELL directory contains the VOCSHELL program.
- The WAV directory contains WAV files.
- The WAVEPOOL directory contains a shareware program and a sample file.

Turn back for more information about Companion Diskette

The SOUND BLASTER™ Book

Revised
Includes information on the new
16 ASP™
Sound Board

The Sound Blaster™ Book explains how to use and explore Sound Blaster™, the popular sound card from Creative Labs.

This book is your guide to the Sound Blaster™, from installation to custom programming. You get an overview of the different Sound Blaster™ versions and the many different commercial, public domain and shareware software products that are available. The *Companion disk* includes valuable and useful programs for your Sound Blaster™.

The Sound Blaster Book is your ticket to the world of multimedia. After chapters dealing with the basics (installation, software, etc.), this book gets down to business: numerous programming examples enable you to transform your PC into a super powerful sound machine.

Other topics covered:

- Tips and tricks for using Sound Blaster™ and Sound Blaster™Pro and the new 16 ASP sound card
- Software support for DOS and Windows MIDI
- Programming the Sound card
- Public domain software and shareware
- Reading Sound Tracker and MIDI files
- Sound Blaster Instrument (SBI) format
- Playing FM, VOC and WAV files
- Onboard Sound Blaster software:
- The Parrot, Dr. Sbaitso and more
- Sound Blaster and Windows 3.1
- Elements of MIDI
- Structure of MOD and MIDI files
- The CT-Voice format (VOC)
- VOC programming
- Sound output under Windows
- FM file playback and much more.

Includes ready-to-use companion diskette

ISBN 1-55755-181-2

$34.95 USA
$44.95 CDN

9 781557 551818

Computer Book Category
IBM/PC & Compatibles: Hardware/Sound
Level: Beginning / Intermediate

Abacus
5370 52nd Street SE • Grand Rapids, MI 49512

Revised

The SOUND BLASTER™ Book

Includes ready-to-use companion diskette

Stolz

181

Abacus